J2EE and .NET
Application Server and Web Services Benchmark

Middleware Company

October 2002

About the Middleware Company

The Middleware Company specializes in advanced enterprise Java technology training and consulting. Founded in 1998 to assist corporations migrating to the Java platform to improve the success of e-Commerce projects, it helps among the world's largest organizations including BEA, Oracle, Cisco, Nextel, MetLife, Sterling Commerce, Standard & Poors, and many others to reduce risks and sustain cost-efficiency by building proficiency in a Java expertise. Instructors are expert architects with deep development experience and strong server-side skills, as well as notable thought leaders in the field. Services include on-site training in Java 2, EJB, J2EE, and XML-Web Services, architecture consulting, and open enrollment courses held worldwide. Courseware is licensed outside North America. The Middleware Company also built and maintains TheServerSide.com, the leading online J2EE community.

Table of Contents

Introduction

In May of 2001 Sun Microsystems® introduced the Java™ Pet Store as a demonstration implementation for J2EE™-based Web applications. According to Sun, the Java Pet Store application illustrates some of the best practices for J2EE development, and is provided as a design pattern for customers to follow when building their own enterprise Web applications. Sun maintains a blueprint series Web site on the Java Pet Store at http://java.sun.com/blueprints/. The Java Pet Store also ships as a primary example application in many leading J2EE application servers.

In November of 2001, Microsoft® announced it had re-implemented the Java Pet Store using the Microsoft .NET Framework and C# to illustrate advantages of the .NET platform over J2EE. The .NET Pet Shop is functionally identical to the Sun Java Pet Store, but is built using Microsoft's promoted architecture for n-tier Web applications based on ASP.NET and the .NET Common Language Runtime (CLR). Microsoft has released benchmark information for the .NET Pet Shop showing the .NET PetShop performance to be significantly better under high user loads than the Java equivalent. The benchmark comparisons are based on comparing the .NET Pet Shop performance to Oracle®-published benchmarks of the Java Pet Store running on Oracle9iAS. The latest comparison based on the .NET PetShop 1.5 can be found at http://www.gotdotnet.com/team/compare/veritest.aspx. Many Java developers as well as Sun, IBM® and Oracle, however, have maintained that the comparisons to date have not been valid, because the Sun blueprint Pet Shop application was not properly optimized for performance and not meant to be benchmarked.

This report contains the results of an extensive new series of benchmarks based on a new implementation of the Java Pet Store, developed by the Middleware Company. The new implementation has been extensively optimized for performance and scalability, and tested in two leading, commercially available J2EE application servers. The new implementation has also been extended with support for distributed XA-compliant transactions across two physical databases. In addition, an XML-based Web service and Web service client have been added to the implementation as well. For benchmark purposes, Microsoft has provided the Middleware Company with a functionally equivalent revised .NET Pet Shop 2.0 application that conforms to the same specification, and has been audited by the Middleware Company. This implementation includes support for distributed transactions via COM+ Serviced .NET components; and the equivalent Web service as the revised J2EE implementation. This report details the results of extensive benchmarks comparing the performance and scalability of the two new implementations.

A Fair Comparison?

From the outset, many Java developers have maintained that the original benchmark comparison was not valid, and that Sun's version was not optimized for either developer productivity or performance. Java developers have pointed out that Microsoft's implementation was specifically optimized for performance, and used a different architecture than the Sun version. For example, Java developers have pointed to Microsoft's use of SQL Server stored procedures in their implementation (the Sun Java version uses dynamic SQL to make database portability easier) as a key difference in the two implementations, and a difference that makes the .NET version faster than the Java counterpart. For its part, Microsoft has maintained that Sun's application was promoted as a "best practice" enterprise design pattern for J2EE, and as such, an examination of the application's scalability is more than justified. Microsoft points out that it has similarly published all of its .NET Pet Shop source code as a "best practice" enterprise design pattern for .NET, and the use of stored procedures is based on the fact many enterprise DBAs prefer to encapsulate queries in stored procedures for control purposes, and many even disallow dynamic SQL in applications. Nevertheless, Java developers on sites such as TheServerSide.com raise legitimate questions; for example:

- Couldn't the J2EE version be re-written and optimized for better performance?
- What would happen to the performance of the .NET version if it did not use stored procedures, and used dynamic SQL as the J2EE version did?
- Were the tests truly comparable? The hardware used in the original comparison, while similar, was not identical, and tests were conducted in separate labs by Oracle and Microsoft, using different versions of the Mercury LoadRunner load test software and different test beds.

A Revised Java Pet Store

Based on this feedback from the Java community, the Middleware Company, as experts with J2EE Web-based application servers and based on feedback from enterprise developer postings on TheServerSide, re-built the Java Pet Store using J2EE and Enterprise Java Beans (EJBs), fully optimizing it for performance and ensuring the new implementation includes only the code required to execute the application, and not code meant simply to demonstrate J2EE features. At the same time, the Middleware Company extended the functionality of the Pet Store application to incorporate important new enterprise functionality including distributed, XA-compliant transactions and XML-based Web Services. The new application has been published on TheServerSide as the Middleware Java Pet Store 2.0. Microsoft similarly updated the .NET Pet Shop to version 2.0, adding the new functionality for distributed transactions and Web Services, as well as changing the .NET application to use all dynamic SQL instead of stored procedures. Microsoft has published the new .NET Pet Shop 2.0 on MSDN as a blueprint for .NET developers to use when building their own n-tier Web applications, and also provided the source code for publication on TheServerSide.

The Middleware Company performed a comprehensive series of benchmarks on the new implementations, using the same hardware for the application server and database

backend system and the same test bed. This report contains the results of these tests as performed and certified by the Middleware Company. The tests, including extensive J2EE application server tuning and optimizations, were performed over a four month period from June – September of 2002.

The basic rules of the benchmark as set forth by the Middleware Company were that:

1. The J2EE and .NET implementations had to be 100% functionally equivalent with no behavioral differences.

2. Both applications had to be created according to best-practice coding standards such that each serves as a valid design pattern that real customers can follow when building their own applications.

3. Each application had to be a logical three-tier implementation, with the use of well-partitioned components to encapsulate middle-tier business and data access logic.

4. The applications had to be designed such that they can each be easily clustered across multiple middle tier application servers for scale-out.

5. The benchmarks had to be run with realistic application server and database deployment settings that reflect a real-world production deployment.

6. All source code, data load and test scripts for the benchmark applications (both .NET and J2EE) have been published on the Serverside.com so that customers can replicate and verify the results. The source code, database schemas, and test scripts used in the benchmark, can be downloaded from: http://www.middleware-company.com/j2eedotnetbench/ .

Extending the Pet Store with New Enterprise Functionality

To make the benchmark more comprehensive, the Pet Store 2.0 application was extended in both J2EE and .NET to include important new enterprise functionality, including:

- **Distributed Transactions.** In the original Sun Java Pet Store, all database transactions are executed across a single database. In the Pet Store 2.0 benchmark, for every order placed, a distributed transaction across two physical databases is performed such that the order information is placed into one database, while the inventory count is updated on a separate database. If one part of the transaction fails, the entire transaction is rolled back. The application and benchmark have been created to compare the performance of J2EE EJBs using JTA transaction services to .NET transactions handled via a .NET Serviced Component utilizing COM+ for transactions.

- **Web Services.** The Pet Store 2.0 now includes a Web Service that provides order information as XML, including all order details and line items for a specific order. The Web Service works over SOAP 1.1. In addition to the Web Service itself, the

Pet Store 2.0 includes a simple client page that invokes the Web Service to display results in a browser.

The Revised Java Pet Store Application

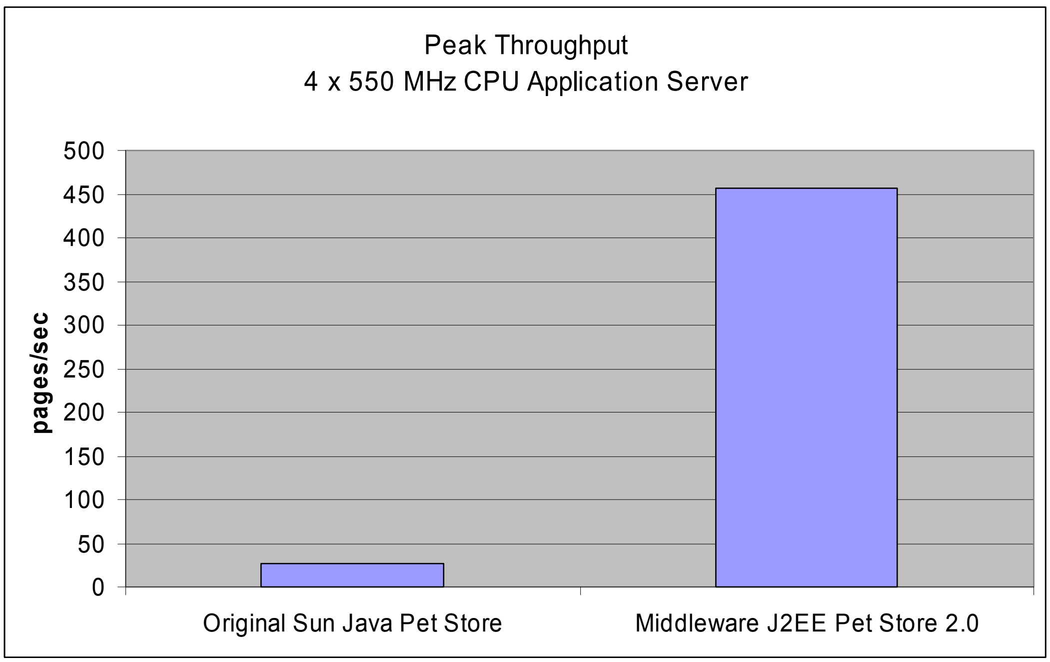
The Java Pet Store application was modified in a number of ways to improve its performance. The revised application remains true to an EJB-based design pattern as promoted by J2EE for scalable, n-tier enterprise applications. However, numerous performance improvements were made to the application to fully optimize it for performance, as detailed below.

- Revision of transaction boundaries: Although most of the original Pet Store application followed the standard access pattern of the application calling a session EJB, which would call all required entity EJBs for a transaction, this was not followed in all cases since it led to some performance degradation in the original implementation. For example, in the case of a table of products contained in a category, three products each with three columns could take nine transactions instead of one. In the revised Middle Java Pet Store 2.0, all transaction boundaries were revised to minimize the number of transactions.
- Implementation of a Read-Mostly pattern for entity EJBs: In the original Pet Store all entity EJBs were developed as read-write. This was simple to develop but inefficient as every operation would cause a database read and also a database write even for static data. To alleviate this problem, the appropriate entity beans were re-implemented using a read mostly pattern. In this, an entity EJB is divided into two; one read-only with its state preserved across transaction boundaries, the other a write entity; both entities share a reference to a common data structure representing their data. The read-only bean interface contains the accessor methods and the write bean the mutator methods; all database read methods are contained in the read-only EJB, e.g. ejbLoad(), and all write methods in the write EJB, e.g. ejbStore(). In addition, isModified() methods were added to all entity EJBs that may write to the database so all unnecessary database updates are avoided.
- Improved use of database indexes: Database indexes were added for commonly used search fields such as product name and item product id's. In addition, a primary key was missing from the original Pet Store's table creation scripts making inventory access unnecessarily slow.
- Removal of excessive JNDI lookup and initialization calls: Each data source and EJB home reference is stored in the application server JNDI service. In the original Pet Store, each time one of these references was used the JNDI initial context was found and then the appropriate reference looked up in JNDI. Both need only be done once. The code was altered so that all unnecessary JNDI calls were avoided.
- One time initialization of dynamic class usage: One of the design patterns used in Pet Store was to show the flexibility of Java by loading classes whose names were not known at compile time. This was accomplished by looking up the name of the class to be used as a string and creating the instance of that class using the

operation forName(). However, the flexibility of the process comes with a large performance cost. In some cases where the class is well known at compile time, this process was removed and the class name hard-coded. In other cases, it was ensured that the lookup/determination of the class to be used only happened once and from then on only new instances of that class created.

As mentioned previously many other changes were made; including the extensive use of JDBC Prepared Statements, significant changes to string handling and others. However, the items detailed above yielded by far the greatest performance gains. Based on comparative performance data for J2EE Application Server A, these optimizations yielded roughly *17 times better performance* over the original Sun implementation of the Java Pet Store. This figure is based on comparative testing on a 4 CPU configuration of the original Sun Java Pet Store modified for functional equivalence (distributed transaction support, data caching behavior) with the new Middleware J2EE Pet Store 2.0. The performance improvement of the new implementation over the old implementation is shown below:

Figure 1. *Comparing the peak throughput of the optimized EJB Java Pet Store 2.0 to the original Sun Java Pet Store.*



The Revised .NET Pet Shop 2.0

The .NET Pet Shop 2.0 architecture, like the Java Pet Store, is logical 3-tier as promoted by Microsoft for building scalable enterprise applications based on .NET. The code base, however, has been streamlined and new performance improvements have been made in some key areas.

The most important changes include:

- Replacement of stored procedures with dynamic SQL for all database access. This removes the contention that performance is achieved through compiled stored procedures at the expense of database portability.
- Incorporation of a distributed transaction on order placement.
- Incorporation of a SOAP-based Web Service to return order details, and calling client page.
- Use of data repeater controls vs. data grid controls for improved performance.
- Incorporation of a simple data cache to cache static read-only product information in a middle tier data cache (note the caching behavior mirrors the behavior of the Java version as achieved with Entity Beans, and is set to refresh every 24 hours).

The new .NET Pet Shop 2.0, along with a complete architecture whitepaper, can be downloaded from http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnbda/html/bdasamppet.asp.

Comparing the New Implementations

The revised Java Pet Store implementation remains true to a Model-View-Controller architecture as published by Sun Microsystems, taking advantage of core J2EE features including Session Beans and Entity Beans. While extensive modifications were made to the application as detailed above to optimize it for performance, the basic design pattern, as recommended by Sun Microsystems and widely practiced for object-oriented development, remains intact. This architecture achieves clean separation of user interface from middle-tier business logic and data access logic, encapsulating backend processing in various EJBs.

The .NET Pet Shop 2.0, similarly, employs a basic n-tier architecture that is also fully object oriented, implemented using C# classes for middle-tier and data access objects. It achieves full separation of UI from middle-tier business logic through the use of ASP.NET Web Forms, which cleanly separates HTML and client-side elements from server-side processing elements using a code-behind model (server-side code is encapsulated in separate files from client-side HTML). The Web Form model takes advantage of ASP.NET server controls provided in the framework for most UI/presentation elements such as lists, grids, etc. In turn, the server controls activate middle-tier objects, written in C# and encapsulated in separate .NET Assemblies (encapsulated, re-usable components), and these middle-tier components access the

database indirectly through a separate set of data access classes that encapsulate the data access logic.

A Word on Stateful/Cached Data

Careful attention was paid to ensuring both implementations behaved the same with respect to how up to date database information presented to the user had to be. Basically, core static product data only had to be refreshed from the database every 24 hours (essentially cached on the middle tier during test runs, although searches still queried the database primary key in order to determine what records to display from the cache), while customer data and user profile data had to be read from the database at every user login, and billing information had to be refreshed from the database as part of the checkout process. In addition, all product inventory quantities had to accurately reflect up to date database information on all application pages presenting this information. The Java version uses Entity Beans to maintain stateful database information; the .NET version uses the .NET middle-tier data cache to store static (read-only) product information, with the cache set to automatically refresh every 24 hours. The configurations achieved identical behavior across the products with respect to data staleness. In the benchmark, dynamic page-level output caching was not used, although both .NET and the J2EE application servers tested support page-level output caching. This decision was made in order to emphasize middle-tier application logic processing performance.

The Extended Benchmark

The benchmark consisted of three suites of distinct tests which provide a fairly comprehensive overview of the performance and scalability characteristics of .NET and J2EE for the two applications. The three benchmark suites included:

1. Web Application Benchmark
2. Distributed Transaction Benchmark 24-hour benchmark
3. Web Services Benchmark

Web Application Benchmark

This benchmark exercises most of the basic functionality of the application, similar but not identical to the workload used in the original Oracle and Microsoft tests. The tests were run to determine a throughput curve as well as measure response times for the application from low to high user loads, pushing each product to high levels of stress to determine the maximum number of users supported in 2, 4 and 8 CPU application server configurations. The test scripts were executed with a 10 second think time (in Oracle's original benchmark, a 20 second think time was used, meaning only ½ the stress was actually placed on the servers at a given user load vs. the new benchmark). Page-level output caching was disabled such that the servers had to process each request received.

Each data point for the different user loads was measured separately with ramp-up, settle-down, data collection and ramp down periods to measure the results accurately (total execution times for each individual data point were over 1 hour). To provide more detailed information, this benchmark suite was performed in two ways:

a. **With no image download**, so that just the base application server engine is utilized. This test indicates how well the product handles application logic including server-side scripting, session state management, object activation, and database connectivity.

b. **With image download**, simulating a browser cache such that each user hitting the site needs to download each unique image exactly once during a script iteration. This test shows how well the application server acts as a combined Web server/application server.

Note that the results of this benchmark cannot be compared to past Oracle and Microsoft published benchmark data for the Pet Shop because the test scripts and think times used are different, and in some areas the functionality of the application is also different than the original Pet Store benchmark.

Distributed Transaction Benchmark 24 Hour Benchmark

This benchmark was designed to stress the distributed transaction processing capability of each product tested. The test script consisted of users logging in, and then proceeding to individually order 100 items. For each item ordered, the checkout process was completed, with the last step in this process being the actual placement of the order that activates a distributed transaction, followed by a logout at the end of the script. Each user therefore completes 100 individual distributed transactions during a user session. The test was run at a user load providing peak throughput for each product for duration of 24 hours to show if this throughput was sustainable.

Price-performance Metric

In addition to the absolute performance metrics, a price/performance metric was calculated for each product based on the 4-CPU configuration in the 24-hour run. This metric is based on $ cost per transaction/sec. The cost is inclusive of the middle-tier hardware cost, the application server software licensing costs (based on the published prices of each product configured for 4-CPUs), plus the purchase price of operating system used for the application server. The price performance metric does not factor in database licensing fees, or ongoing support/maintenance contracts.

Web Service Benchmark

This benchmark measured the performance characteristics of Web services for both J2EE and .NET. The test included two basic configurations:

a. **Direct activation of the Web Service** by 100 distributed physical computers each simulating multiple users making direct SOAP calls to activate the Web Service. This benchmark tests the ability of the application server to handle incoming SOAP requests and act as a Web Service provider.

b. **Remote activation of the Web Service** by the application server via a proxy object. In this configuration, two physical application servers were

configured, one to act as the remote Web Service provider (as in test a), the other to act as a Web Service client. The 100 physical client computers generating load make HTTP/HTML requests to the application server computer, which in turn makes SOAP requests the Web Service provider machine. This test is designed to test the application servers Web Service client performance and performance when making remote SOAP-based object activations.

The Test Lab and Test Software

Mercury LoadRunner 7.5 was setup and configured in the lab by Mercury Interactive representatives. The tests were run in a large-scale test lab that consisted of 100 physical client machines capable of generating very high concurrent user loads while ensuring the clients were not a bottleneck in the system. A CISCO gigabit backbone was used, with each server configured with two gigabit network cards each talking to a subset of 50 clients. This setup was designed to ensure the network was never a bottleneck during testing. Two separate database servers were also configured on the gigabit network. The application servers tested were run on Compaq ProLiant 8500 Servers, configured with 2, 4 and 8 550 MHz CPUs, and 2 GB RAM (2, 4 CPU setups) and 4 GB RAM (8 CPU setup). The two databases were also Compaq ProLiant 8500 servers, each with 8 550 MHz CPUs, 3 GB RAM, and fiber optic controllers attached to fast Compaq RAID storage arrays. Database, client, and network utilization were monitored during testing to ensure these never became a bottleneck. In all cases for J2EE and .NET, the tests accurately report the capability of the application server software itself for the applications tested.

Figure 2. *Lab Configuration for Base Web Application Benchmark, 24 Hour Distributed Transaction Benchmark and Web Services Direct SOAP Activation Benchmark*

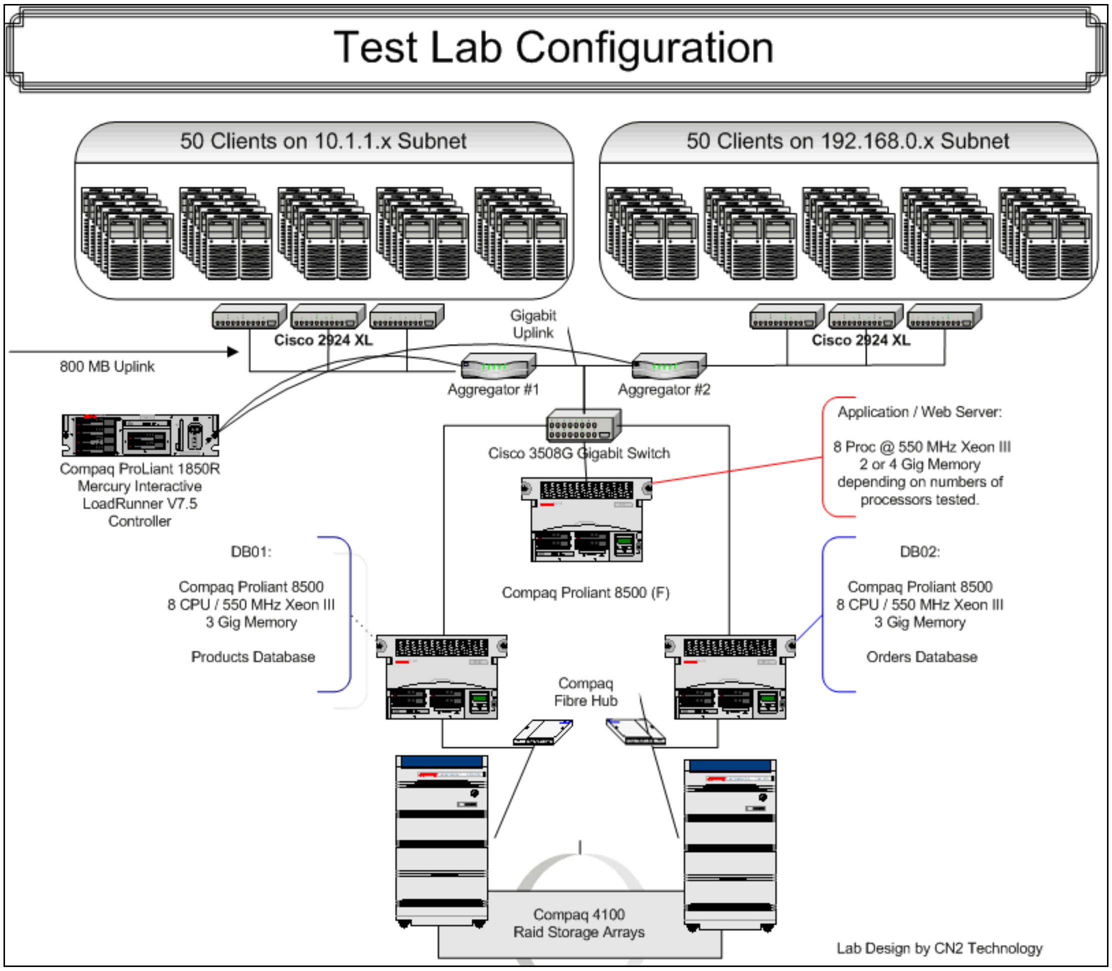
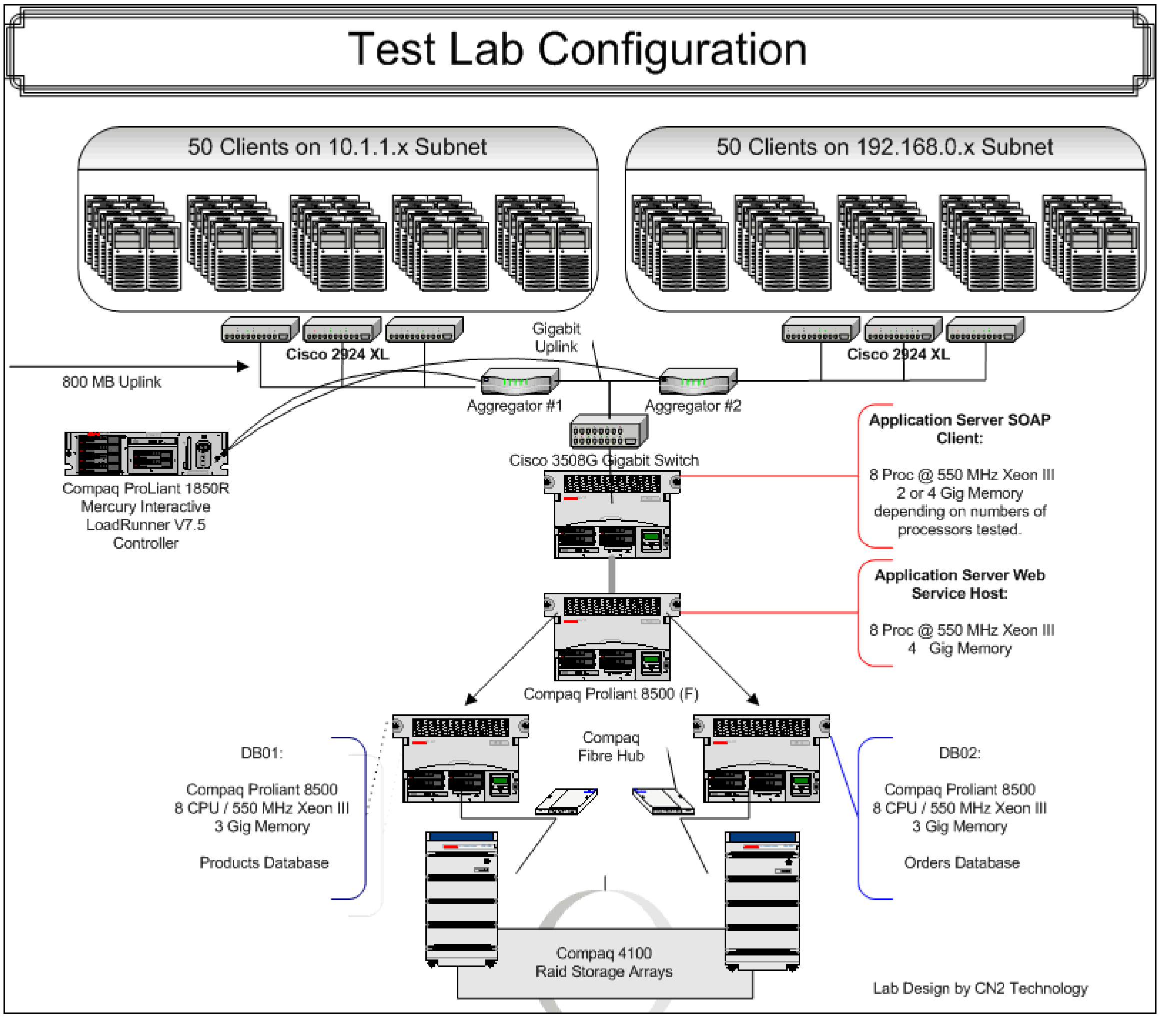


Figure 3. *Lab Configuration for Remote/Proxy Web Services Benchmark with a remote SOAP call between two application servers.*



Products Tested

Microsoft .NET

On the .NET side, the Middleware Company tested the commercially released .NET Framework 1.0 (SP2) running on Windows 2000 as well as the upcoming .NET Framework 1.1 RC build 4322.508 on RC Build 3681 of Windows.NET Server 2003. Microsoft SQL Server 2000 was the database used as the backend in these tests. Precise configuration optimizations for .NET 1.0 and .NET 1.1 are documented in Appendices 7-8.

Time spent on .NET tuning and configuration optimizations prior to testing

After developing the .NET Pet Shop 2.0 implementation (the application was actually developed by a California-based Microsoft Solution Provider), a single Microsoft employee spent roughly 2 weeks on trial runs and final optimizations including configuration settings prior to the Middleware testing (total 2 man-weeks).

.NET tuning and configuration optimizations

The optimizations made were minimal changes from default .NET Framework settings. Optimizations included:

- Ensuring the database was tuned properly and had enough storage allocated for the 24-hour transaction-heavy run.

- A change to the base ASP.NET settings to enable Forms authentication vs. Windows-based authentication since the application itself is a thin-client application that can be accessed via any client platform with a standard browser (authentication is based on a database of registered usernames and passwords).

- Tuning ASP.NET worker and IO threads down from 25 (default on Windows 2000) to 20; and slightly increasing the number of network connections allowed per client IP on the Web Services test runs to ensure the server allowed each load generating client to activate enough concurrent network connections to the .NET application server.

- On the Windows 2000 tests, the ASP.NET process model was set to allow the .NET application to run in the same process space as the IIS Web Server (a similar setup to using the built-in HTTP listener for one of the J2EE app servers tested).

- On the Windows.NET Server 2003 tests, the new IIS 6.0 process model (default setting) was used since this provided better performance than Windows 2000 but allowed the application to run in a dedicated server process (application pool) separate from the HTTP Server.

- Under Windows.NET Server, two worker processes were enabled in the IIS Service Manager for the 4 and 8 CPU systems. ASP.NET worker processes are automatically spawned and monitored by ASP.NET on Windows.NET Server, and are similar in concept to running application server "clones" with J2EE applications.

J2EE

On the J2EE side, the Middleware Company performed tests using two leading J2EE application servers, identified in this report as J2EE Application Server A and J2EE Application Server B. License restrictions prevent the Middleware Company from disclosing the names of the J2EE application servers tested. However, the products chosen represent widely used, state-of-the art, commercially marketed enterprise J2EE application servers. The J2EE application servers were tested with Oracle 9i backend databases, since this database is one of the primary recommended databases for each product, with full support for J2EE/JDBC application server features.

Both application servers were tested on both RedHat Linux 7.2 and on Windows 2000 Advanced Server (SP2). Middleware performed a comparison of each on both operating systems, and used the operating system that provided the best performance with that application server for the final, published test runs. For J2EE Application Server A, Windows 2000 was chosen since the application server performed noticeably better on Windows 2000 than Linux 7.2. For J2EE Application Server B, both Windows 2000 and Linux 7.2 provided comparable performance, but Windows 2000 was again chosen primarily because it was easier to monitor the performance characteristics of the computer under load without effecting performance by using the built-in Windows 2000 Performance Monitor. Precise configuration optimizations J2EE Application Server A and J2EE Application Server B are documented in Appendices 5-6.

Time spent on J2EE tuning and configuration optimizations prior to testing

After development of the J2EE optimized application, two experienced Middleware developers (full time) spent roughly 5 weeks on tuning and configuration optimizations for J2EE Application Server A before taking final data points (10 man-weeks). Another 5 weeks was spent on tuning and configuration optimizations for J2EE Application Server B (10 man-weeks). The process of tuning and optimizing the configurations for both J2EE application servers was non-trivial as detailed below.

J2EE tuning and configuration optimizations

The J2EE application was extensively tuned independently for each application server product tested by Middleware experts according to best-practice vendor guidelines. This included testing each product with a wide variety of Java Runtime Environment (JRE) and database drivers in order to determine the optimal combination; as well as base application server configuration optimizations such as heap sizes, bean caches and thread settings.

In tuning the two application servers, the first points that were considered were the choice of Java Runtime Environment (JRE) and JDBC driver to be used. In this, four JREs were tested: SunSoft 1.3 and 1.4, JRockit 3.1 and IBM 1.3.

On application server A, SunSoft 1.4 was found to be by far the fastest for this application, offering up to 50% better throughput in the benchmark than SunSoft 1.3. However, because of compatibility reasons SunSoft 1.4, while it could be used for the Web application and distributed transaction benchmarks, could not be used on J2EE Application Server A for the Web service testing so the JRockit JRE was used in this case. One important factor in the performance of these JREs is the availability of concurrent garbage collection. If this feature is not available, the pauses for garbage collection become excessive and can lead to "connection refused" errors under high load as application server processing stops during garbage collections. J2EE Application Server A benefited from the fact that SunSoft JVM 1.4, which provided much optimized garbage collection over 1.3, worked with the product (except in the Web Service test).

J2EE Application Server B was not compatible with the SunSoft JVM 1.4 yet, so the IBM JRE 1.3 was used. This JVM provided the most optimal performance with J2EE Application Server B, although the garbage collection was still lacking when compared to that provided by the SunSoft JVM 1.4. It was found that with Application Server B, better and more reliable performance was obtained by using the application server's front end Web server (running as a separate process) to throttle incoming requests to the application server itself. This is a practice that is recommended by Vendor B for their application server. By properly throttling the number of concurrent requests coming into the application server, the server ran faster and more reliably under load, and the impact of garbage collections is minimized. The use of the application server's Web server (vs. its built-in, in-process http listener) did require additional configuration and tuning to determine the optimal settings. The key configuration change with the Web server was limited to only one setting which limited the number of concurrent connections/threads the Web server would initiate with the application server. This setting was set relatively low (50 threads) in order to queue incoming requests on the Web server, while keeping the queue very small in the application server engine. The setting had to be tuned in connection with thread tuning on the application server itself, a process that proved tricky. But ultimately this setup ensured that the application server engine itself could never become overwhelmed by a high number of incoming requests, and hence would run more reliably. If the Web server was configured to allow too many user connections, throughput would drop dramatically as the application server became unstable. Even with this extensive tuning, J2EE Application Server B did not perform nearly as well as J2EE Application Server A in the Web Application and Distributed Transaction benchmarks. Poor performance with the JTA distributed transaction coordination (which is activated in both of these benchmark tests) was a large contributing factor; as was the required use of the 1.3 vs. 1.4 JRE.

The selection of JDBC drivers was based on a number of criteria:

- Availability of features: XA transaction support and scrollable result sets.
- Overall performance.

- Availability of type 2 (thick) and type 4 (thin) versions of the driver.

Four drivers were tested: The database vendor drivers (Oracle), the application server vendor drivers and two leading third party driver vendors. After first eliminating those which did not have the required features, the remaining drivers were tested for performance and stability under load.

After deciding upon the JRE and JDBC driver to use, the application servers were tuned for both the application server parameters and the Java runtime parameters. In addition to the tuning of an individual server instance, greatly improved performance was also achieved by running more than one server process on the application server machine using DNS to load balance between the server instances. In large part this configuration was optimal for both application servers because it allowed for utilization of the full amount of memory on the server, but each clone had to individually garbage collect less memory, meaning the major periodic garbage collections would occur much faster minimizing the corresponding halts in processing.

Web Application Benchmark Results

This benchmark contained a 50/50 mix of two scripts one of which simulated a user simply browsing the site; the other simulating a user looking for and purchasing items from the site. In the browse-only script the user would:

1. Go to the site home page
2. Perform three (3) free text searches on 1,000 products, clicking the next button on each search result page.
3. Examine the products in a category three (3) times.
4. Examine the details of a particular product item three (3) times (this includes a real-time read of product inventory counts).

In the browse and purchase case the simulated user would:

1. Perform two (2) free text searches on 1,000 products, clicking the next button on each search result page.
2. Sign on to the site using a random user id from one of the 100,000 available users.
3. Add two (2) random items, from 50,000, to the shopping cart.
4. Checkout, verify the account information and place the order for the contents of the cart.
5. Half of the time the user would signout at this time and half of the time not such that 50% of sessions were closed, while 50% were abandoned such that the application server would need to handle closing them down, with a session timeout set for 10 minutes of inactivity. This was felt to be a reasonable simulation of the actual activity on a real e-commerce site.
6. Perform one last free text search after purchase.

Testing Methodology

The tests included two completely separate runs for each product, one with **image download turned off**, and one with **image download turned on** (with a browser cache on for the clients). This was done to show a more complete picture of application server performance inclusive of both raw backend processing usage scenarios where the application server itself is not responsible for serving Pet Store Web site images; and general-purpose deployment scenarios where the application server will be expected to both process application logic and serve images down to the browsers. Note that the .NET Pet Shop and the Middleware J2EE Pet Store use different images in their implementations. Hence, for the image runs, the .NET Pet Shop was modified to load the J2EE Pet Store images, such that the number of images and image size was the same for each tested product.

Also, in the image download case with simulated browser cache, the benchmark settings were configured such that images would only be downloaded once for each user (for each

unique image) during a user iteration, similar to a real-world setting where browser cache enables images to be cached on the client for performance reasons. Hence, common images across pages, like navigation elements, were only downloaded once per user session even as a given simulated user visits multiple pages during script execution. However, each new simulated user was configured with its own clean browser cache such that each simulated user had to populate its image cache independently.

The tests were performed on 2, 4 and 8 CPU configurations to illustrate vertical scaling of the application servers as CPUs are added to the system. A data run consisted of a ramp up, settle down, data collection and ramp down period for each individual data point. Data collection only began once steady state had been achieved by the system, with total test runs of roughly one hour each. For a given product, data points were taken only up to the user load at which response times were 3 seconds or more, since testing beyond these loads for a given product typically returned errors. Hence, more data points were collected for those products that scaled to handle higher loads.

Web Application Benchmark Results - Image Download Off

Figure 4. *Peak throughput.*

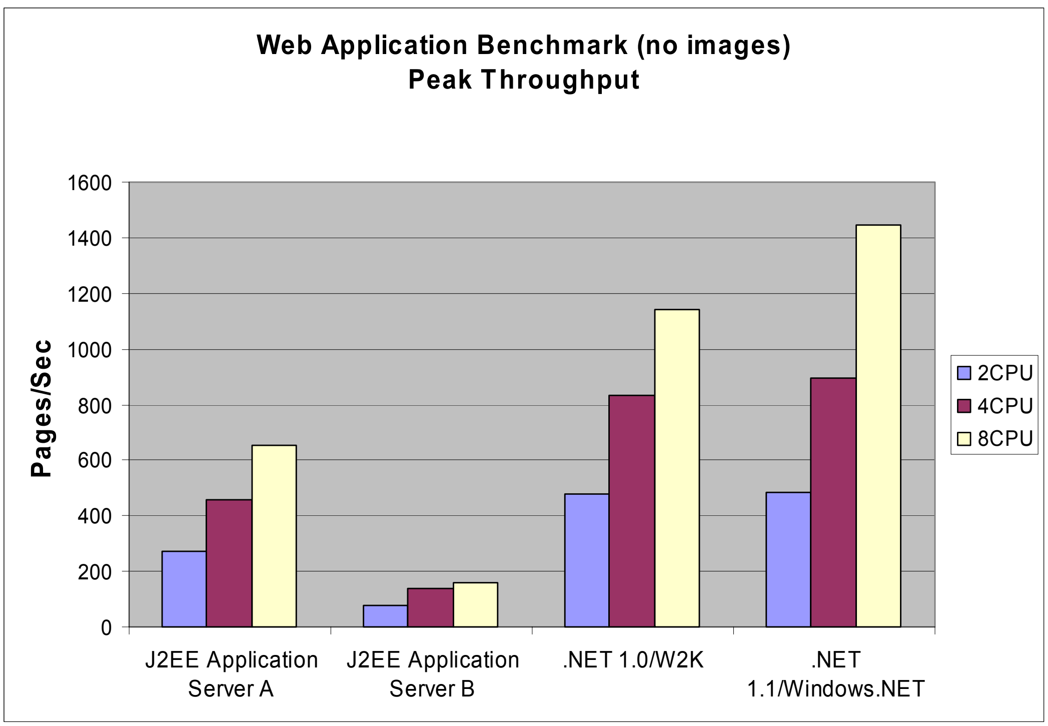


Figure 5. *Maximum supported user load @ 3 second average page response threshold.*

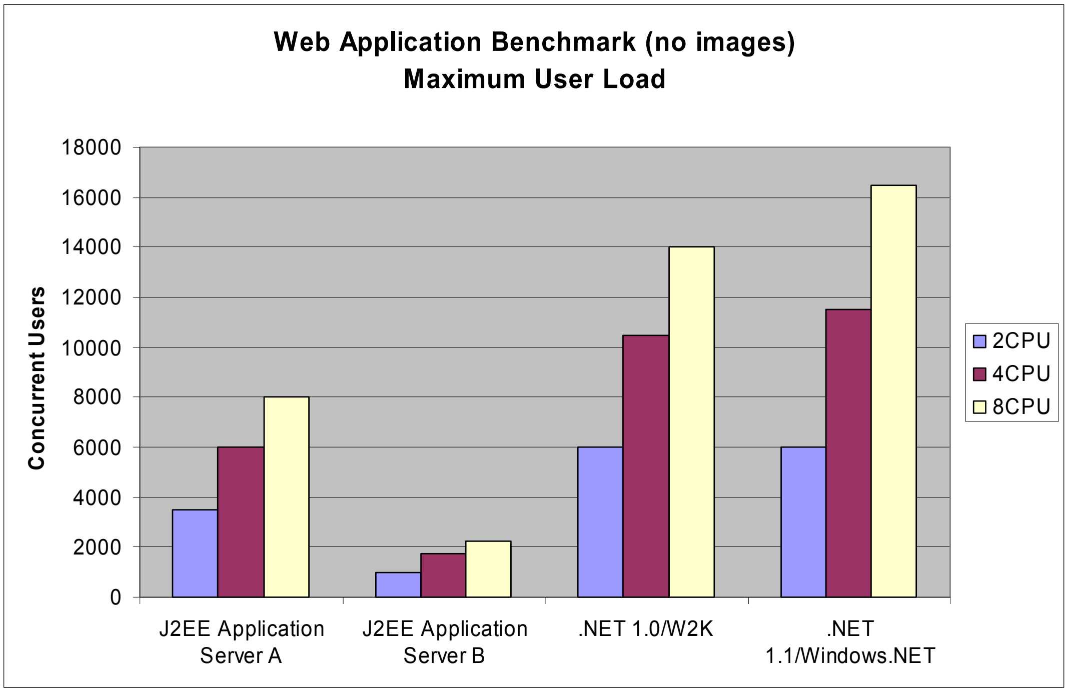


Figure 6. *Throughput curves 2 CPU Application Server.*

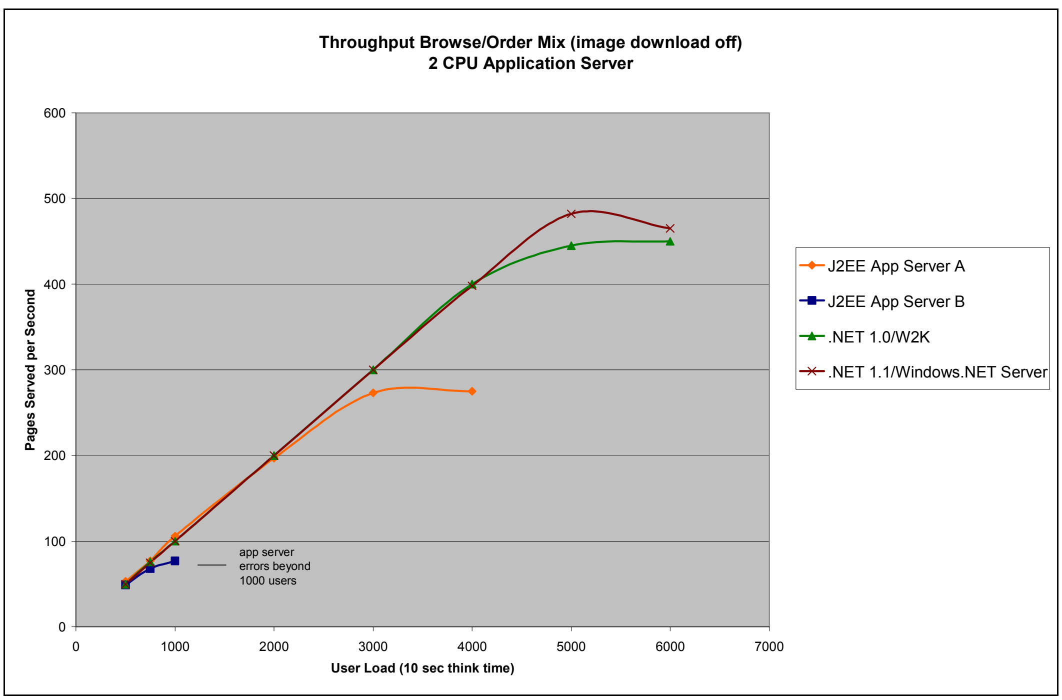


Figure 7. *Throughput curves 4 CPU Application Server.*

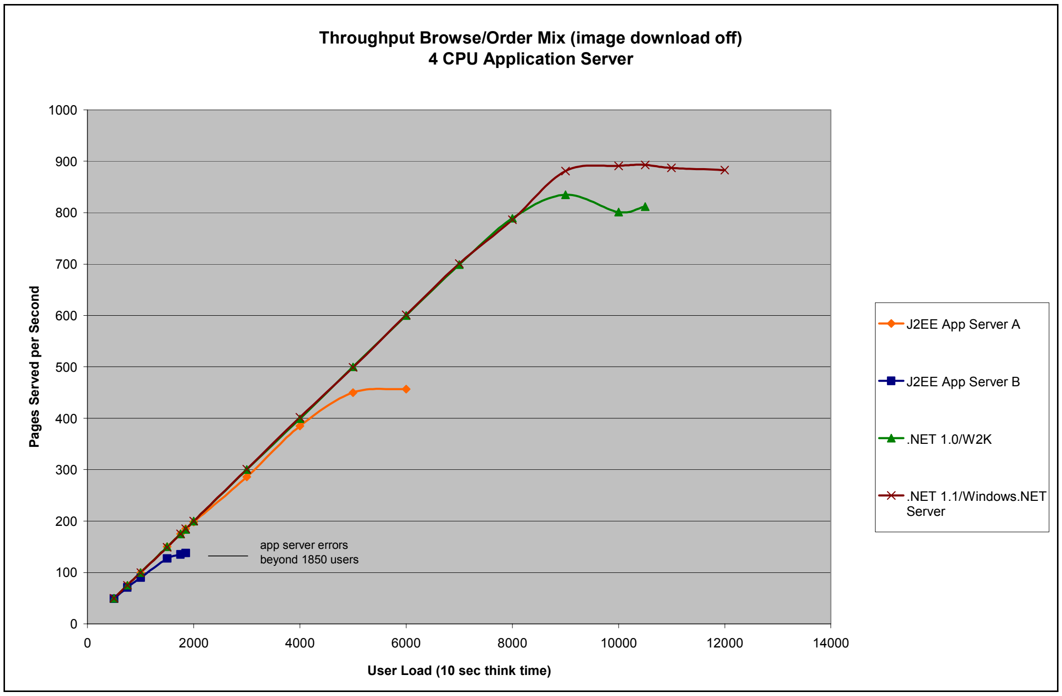
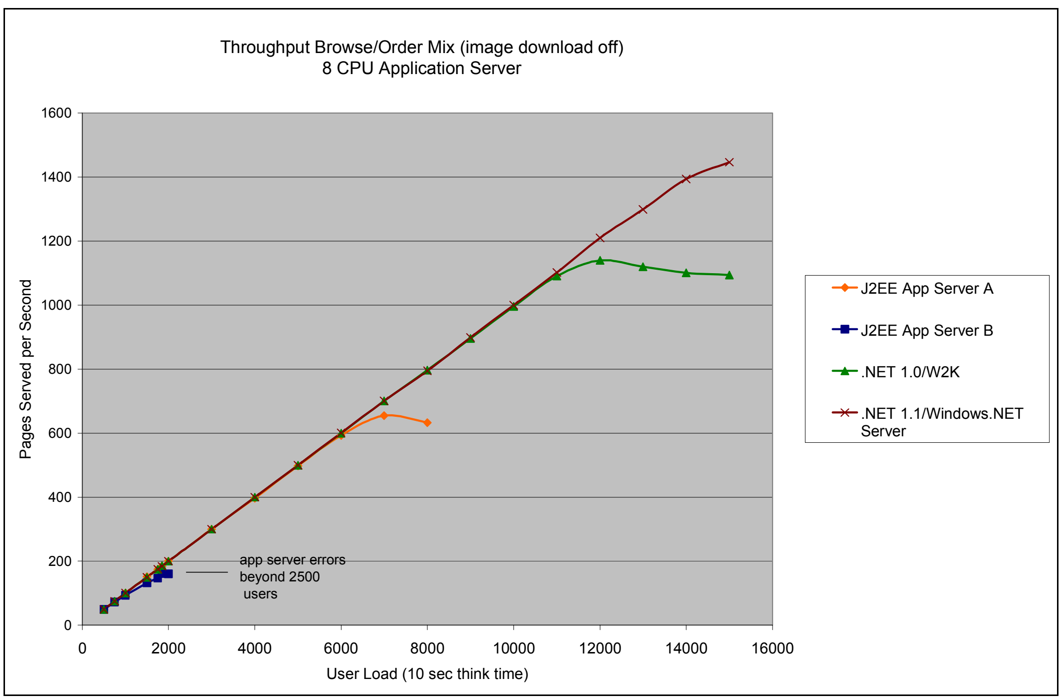


Figure 8. *Throughput curves 8 CPU Application Server.*



Web Application Benchmark Results - Image Download On

Figure 9. *Peak Throughput.*

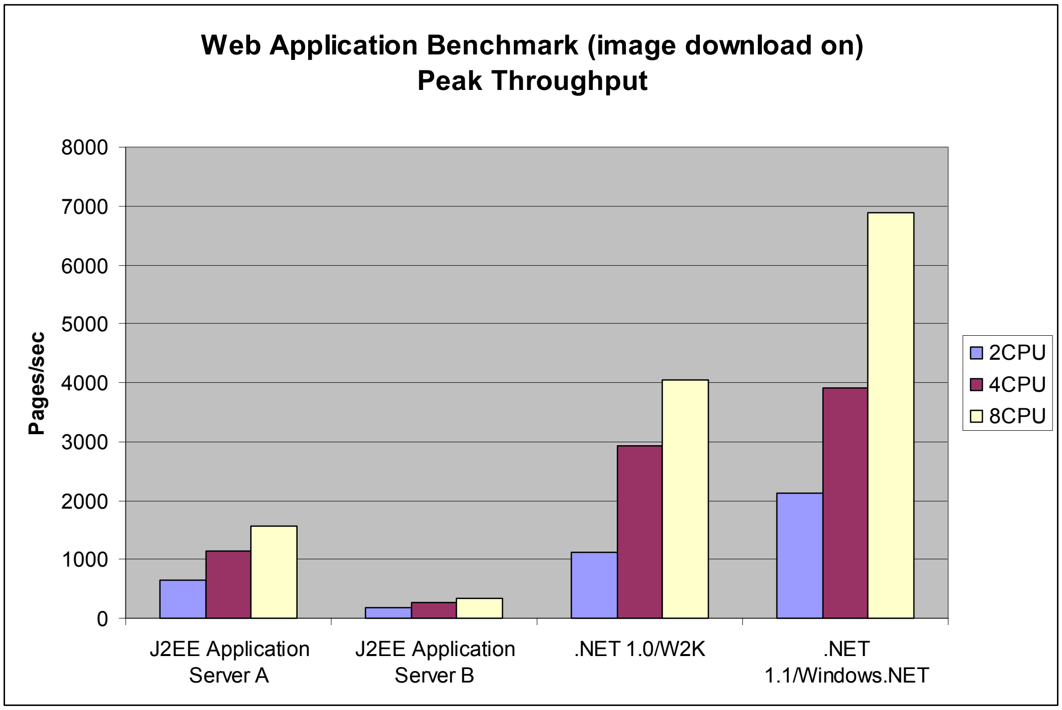


Figure 10. *Maximum supported user load @ 3 second average page response threshold.*

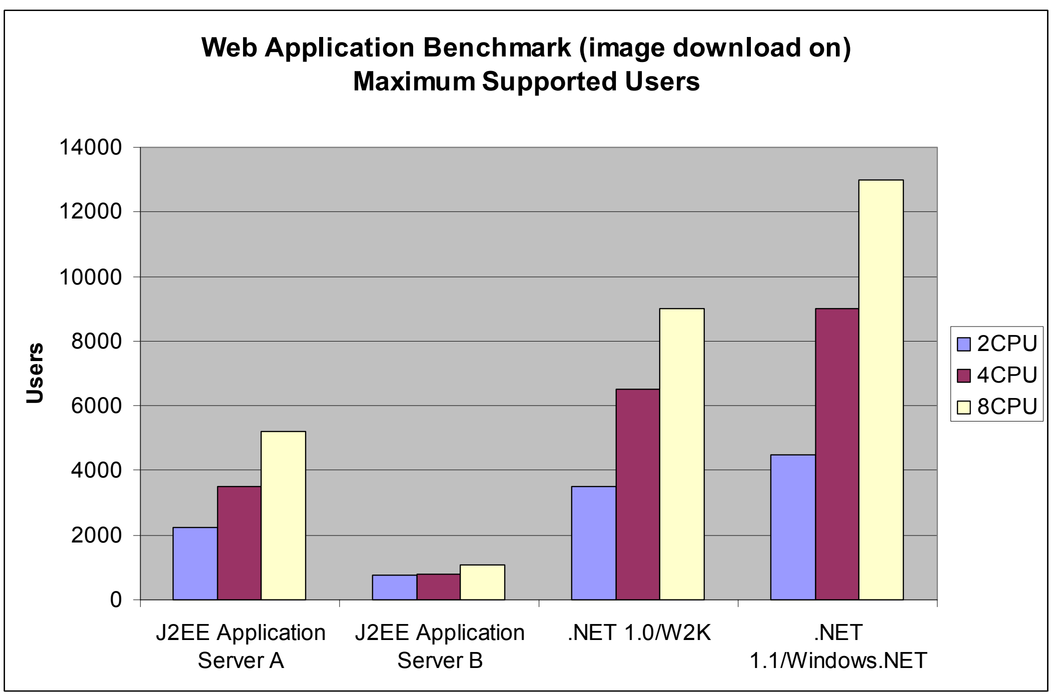


Figure 11. *Throughput curves 2 CPU Application Server.*

Figure 12. *Throughput curves 4 CPU Application Server.*

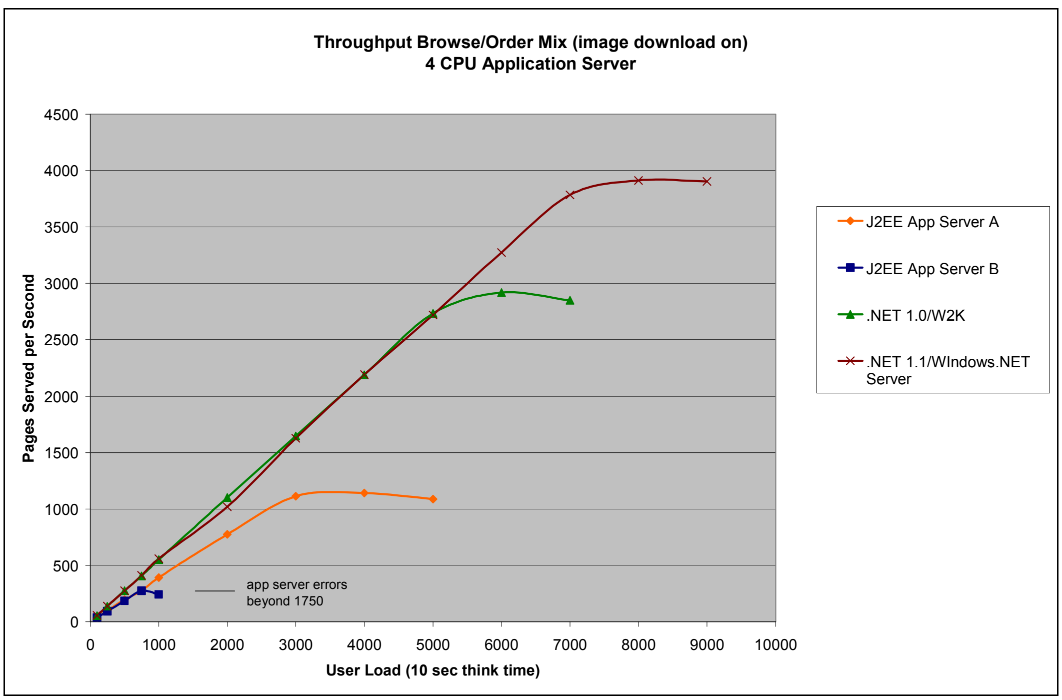
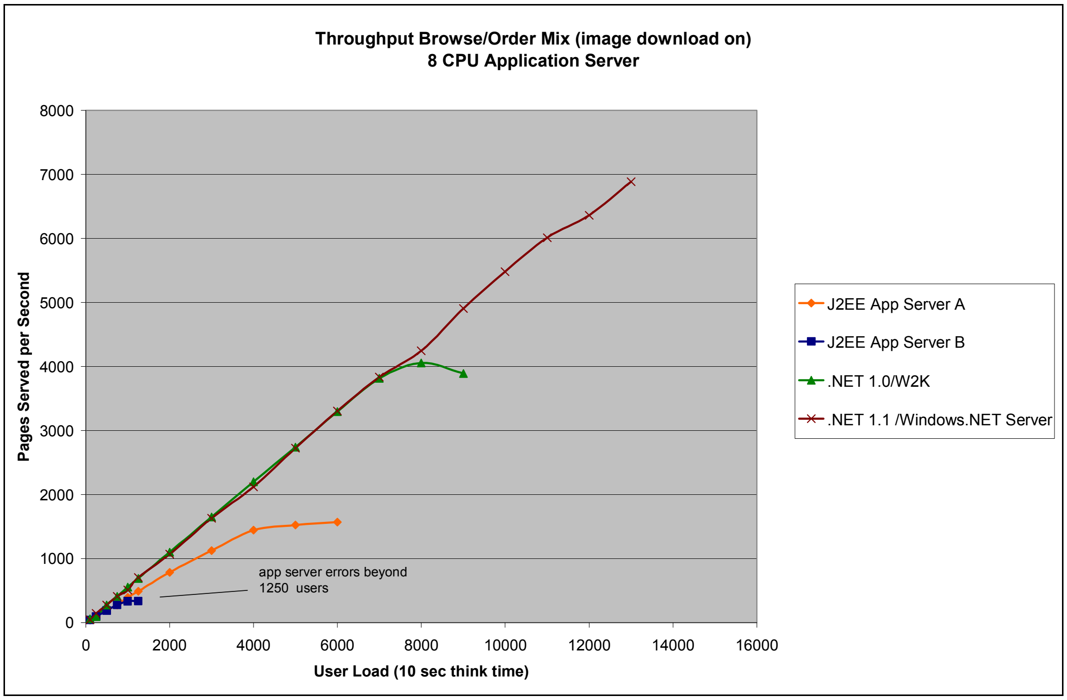


Figure 13. *Throughput curves 8 CPU Application Server.*



Distributed Transaction Benchmark 24 Hour Benchmark

In the distributed transaction benchmark, the script was designed to thoroughly test the application server ability to handle high loads of distributed transactions; in this case, the writing of orders. The script used simulated users doing the following:

1. Sign on to the site using a random user id from one of the 100,000 available users.
2. Adding a random item, from 50,000, to the cart and immediately ordering the item, performed 100 times each after login. The checkout process, performed 100 times for each login, includes verifying the account information and placing the order for the contents of the cart (a distributed transaction).
3. Signout.

Note that the ratio of actual distributed transactions to total server requests in this test was 1:5, since the checkout process in the applications involves a 5-step, 5-page order confirmation/order placement process.

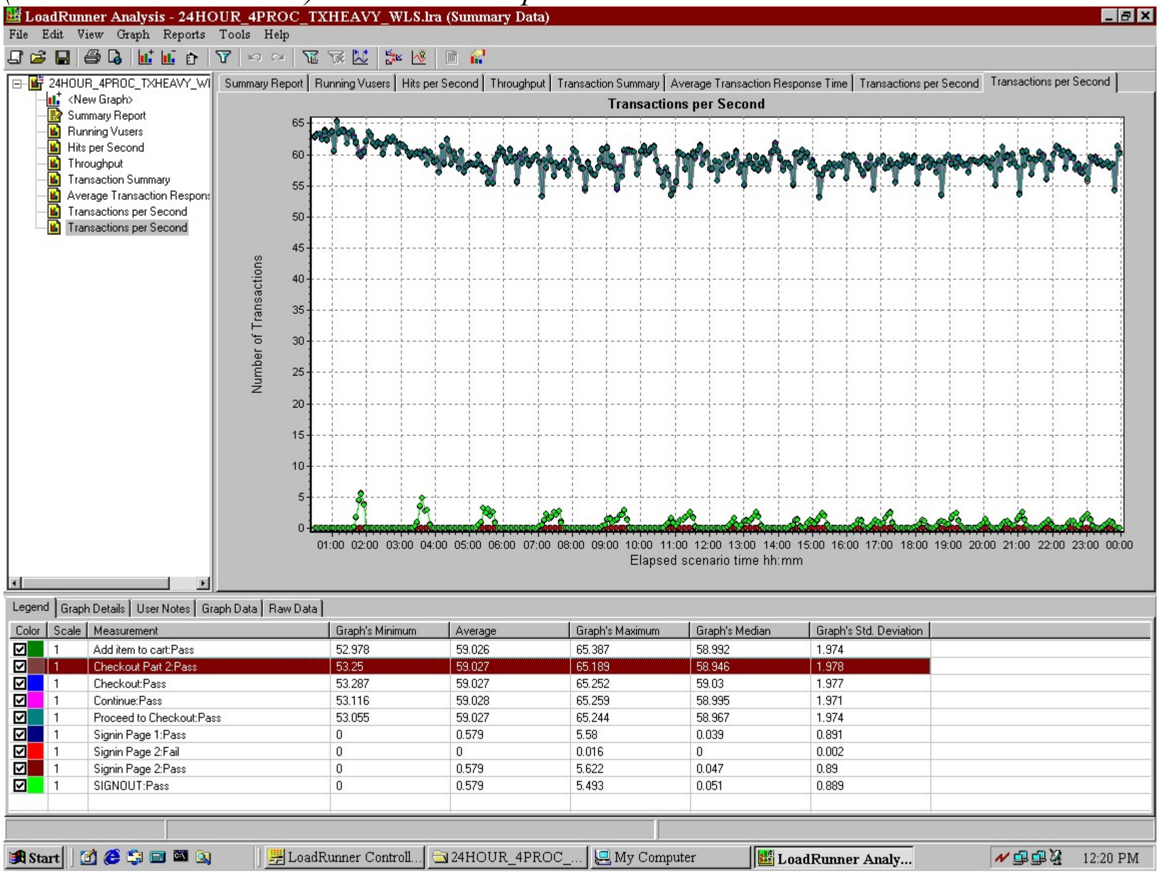
The distributed transaction benchmark was run on a 4-CPU application server at a user load that achieved peak, sustainable throughput for each application server product tested, as depicted below:

Figure 14. *Distributed transaction benchmark results.*

Application Server	Peak Throughput User Load	Distributed Transaction Orders Placed per Second	Price/Performance[1]
J2EE Application Server A	4,000	59 TPS	**$1,305**
J2EE Application Server B	1,000	18 TPS[2] (unable to sustain, see footnote)	**$4,722 (see note 2)**
.NET 1.0/WINDOWS 2000	4,000	79 TPS	**$468**
.NET 1.1/Windows.NET Server 2003	6000	117 TPS	**$316[3]**

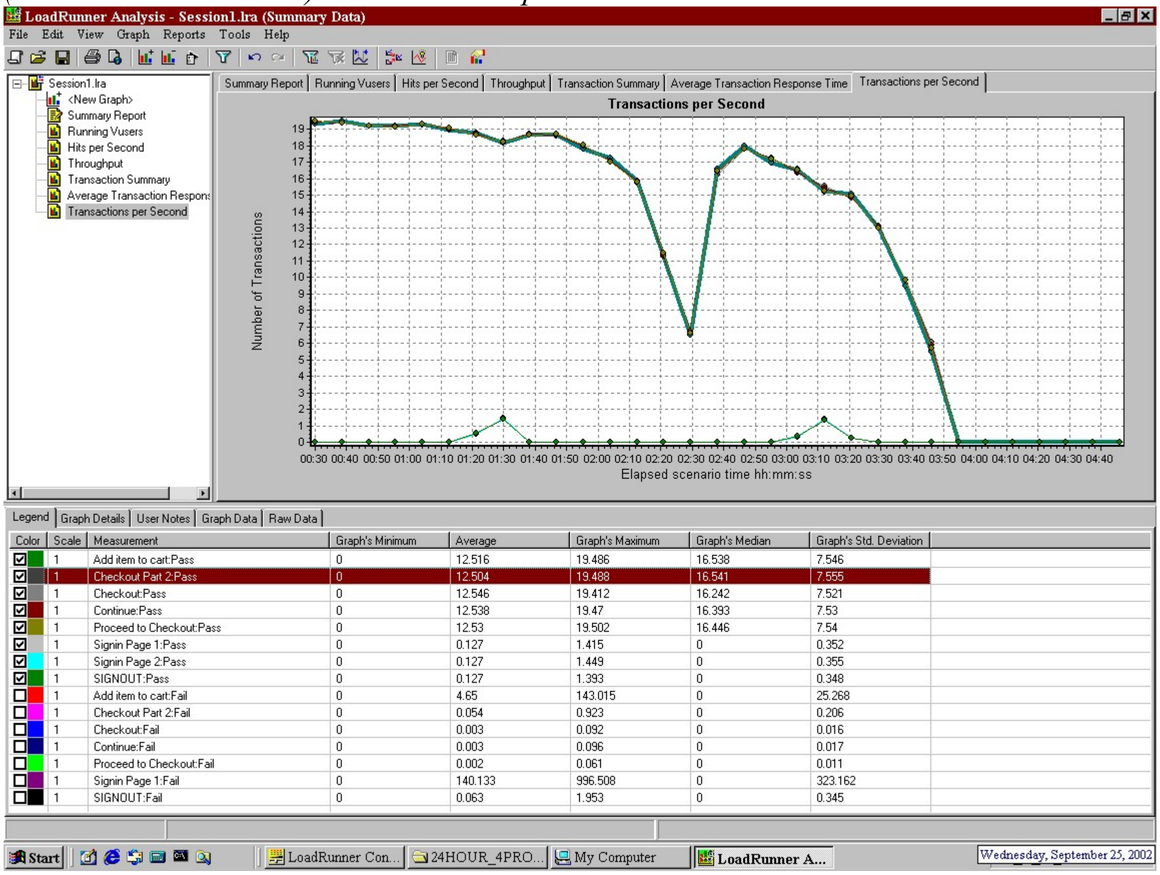
[1] See Appendix 2 for total hardware/software cost breakdown of Application Server systems.
[2] J2EE Application Server B was unable to sustain peak throughput for more than 2 hours on the distributed transaction benchmark. It was unable to sustain any throughput beyond 4 hours, destabilizing over this period of time to the point of failure.
[3] Estimate. Based on Windows 2000 Advanced Server pricing, Windows.NET Server 2003 pricing not yet published.

Figure 15. *Mercury LoadRunner results analysis for **J2EE Application Server A** showing throughput over 24 hours. This chart depicts the transactions per second for the individual Web requests during the 24 hour period. Note that the actual order placement (distributed transaction) is the checkout part 2 transaction.*



Notes: Throughput declined slightly over the first six hours, then was sustained for an average TPS on the order placement (distributed transaction) over the full 24 hours of 59 orders/second.

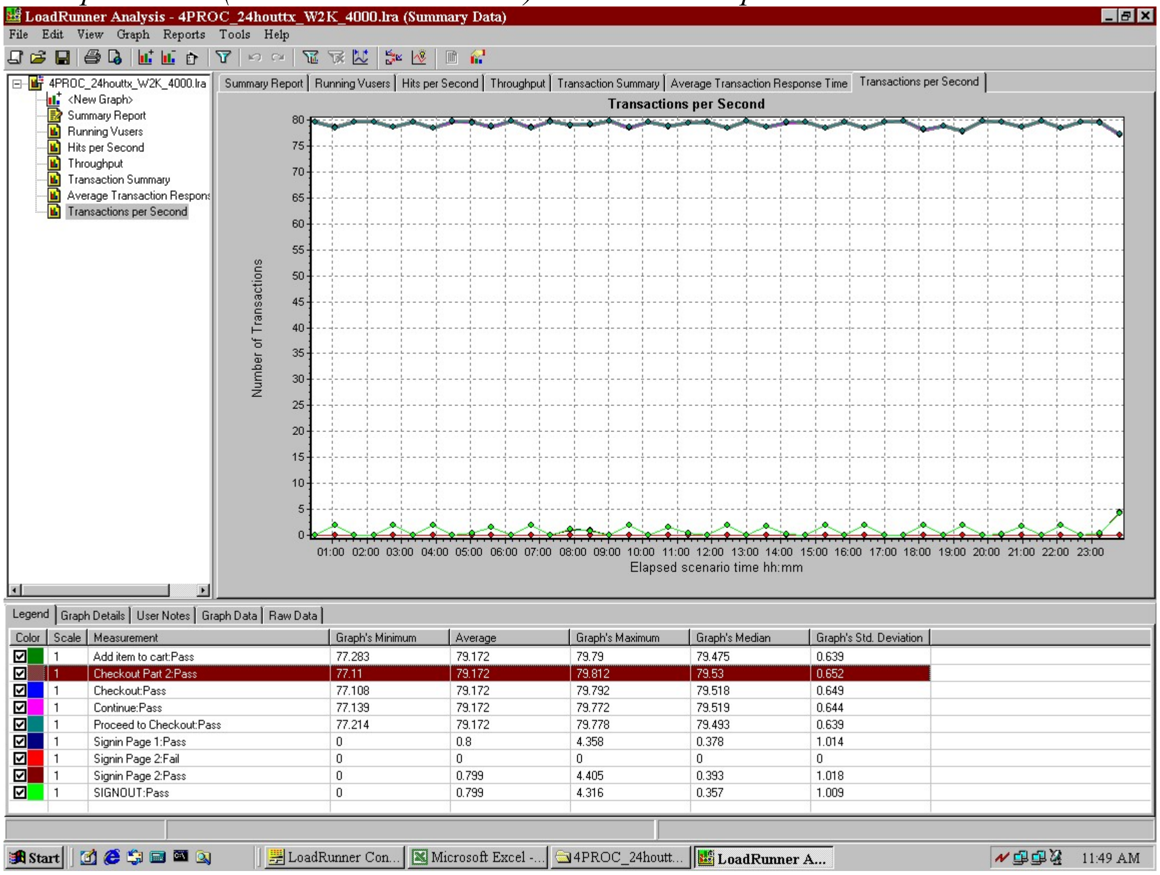
Figure 16. *Mercury LoadRunner results analysis for* ***J2EE Application Server B*** *showing throughput over 24 hours. This chart depicts the transactions per second for the individual Web requests during the 24 hour period. Note that the actual order placement (distributed transaction) is the checkout part 2 transaction.*



Color	Scale	Measurement	Graph's Minimum	Average	Graph's Maximum	Graph's Median	Graph's Std. Deviation
☑	1	Add item to cart:Pass	0	12.516	19.486	16.538	7.546
☑	1	Checkout Part 2:Pass	0	12.504	19.488	16.541	7.555
☑	1	Checkout:Pass	0	12.546	19.412	16.242	7.521
☑	1	Continue:Pass	0	12.538	19.47	16.393	7.53
☑	1	Proceed to Checkout:Pass	0	12.53	19.502	16.446	7.54
☑	1	Signin Page 1:Pass	0	0.127	1.415	0	0.352
☑	1	Signin Page 2:Pass	0	0.127	1.449	0	0.355
☑	1	SIGNOUT:Pass	0	0.127	1.393	0	0.348
☐	1	Add item to cart:Fail	0	4.65	143.015	0	25.268
☐	1	Checkout Part 2:Fail	0	0.054	0.923	0	0.206
☐	1	Checkout:Fail	0	0.003	0.092	0	0.016
☐	1	Continue:Fail	0	0.003	0.096	0	0.017
☐	1	Proceed to Checkout:Fail	0	0.002	0.061	0	0.011
☐	1	Signin Page 1:Fail	0	140.133	996.508	0	323.162
☐	1	SIGNOUT:Fail	0	0.063	1.953	0	0.345

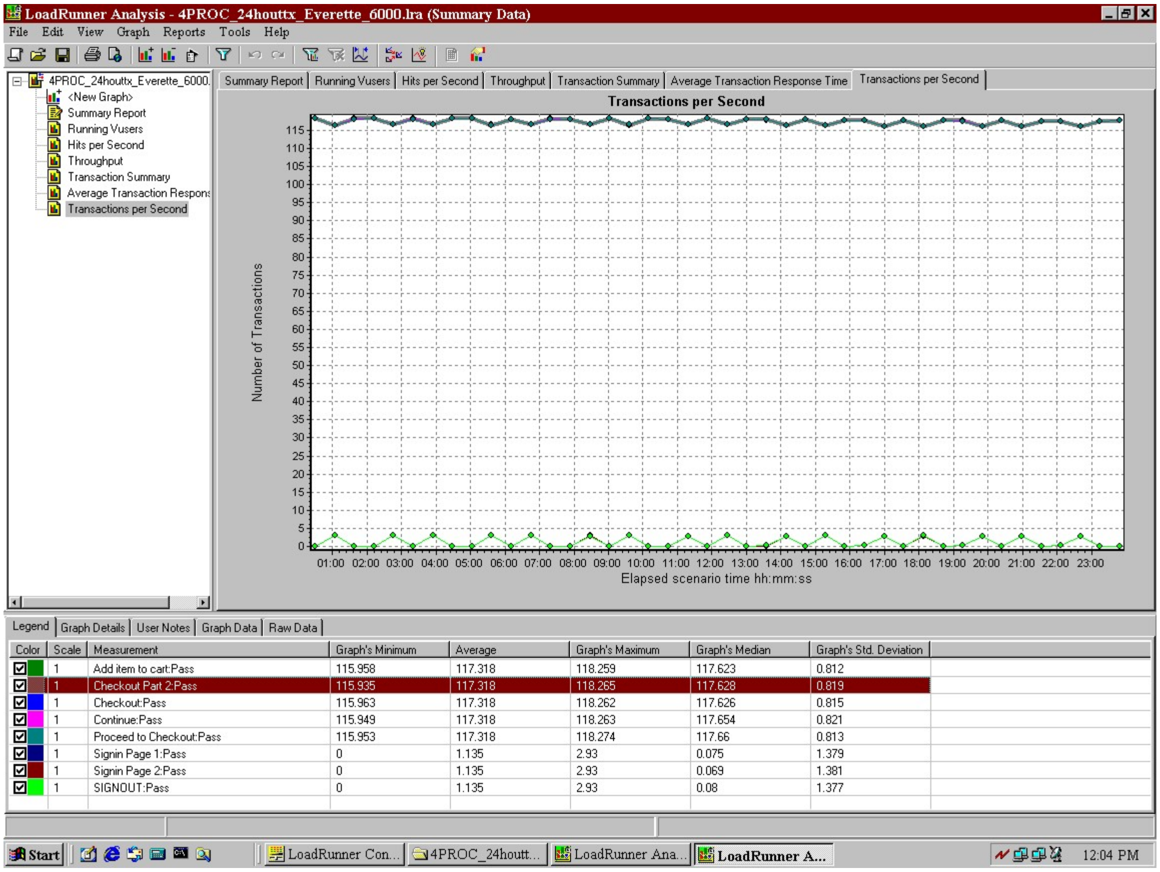
Notes: Peak throughput on the distributed transaction benchmark was not sustainable beyond 2 hours. With over 40 different attempts with various adjustments to tuning parameters, J2EE Application Server B had significant difficulty with distributed XA-compliant transactions against the database. The run depicted above is typical, with slightly falling TPS, followed by a dramatic decline, a recovery, followed by an application server failure such that no requests are successfully processed beyond 4 hours.

Figure 17. *Mercury LoadRunner results analysis for* ***Microsoft .NET 1.0/Windows 2000 Server*** *showing throughput over 24 hours. This chart depicts the transactions per second for the individual Web requests during the 24 hour period. Note that the actual order placement (distributed transaction) is the checkout part 2 transaction.*



Notes: Throughput was steady throughout the 24 hour period for an average TPS on the order placement (distributed transaction) over the full 24 hours of 79 orders/second.

Figure 18. *Mercury LoadRunner results analysis for **Microsoft .NET 1.1/Windows.NET Server 2003** showing throughput over 24 hours. This chart depicts the transactions per second for the individual Web requests during the 24 hour period. Note that the actual order placement (distributed transaction) is the checkout part 2 transaction.*



Notes: Throughput was steady throughout the 24 hour period for an average TPS on the order placement (distributed transaction) over the full 24 hours of 117 orders/second.

Web Services Benchmark

Since Web Services have been extensively promoted as the new frontier of application servers, the new Pet Store benchmark was extended to provide a look at the performance of each application server for a prototypical Web Service. The Web Service itself was specified such that it would provide a realistic test case to illustrate application server performance inclusive of base object activation over SOAP, as well as serialization of simple and complex data types/objects into XML. The functionality of the Web Service in the Pet Store is to take as a single input parameter an OrderId, and then to perform a database lookup of that order, returning an Order object as XML to the calling application. The Order object itself contains both simple data types (string, integers, decimals) as well as an array (representing individual order detail lineitems).

Figure 19. *Basic XML Schema for Order Object Returned from Web Service Call Based on SOAP 1.1.*

```
<GetOrderResult>
        <orderId>int</orderId>
        <date>dateTime</date>
        <userId>string</userId>
        <cardType>string</cardType>
        <cardNumber>string</cardNumber>
        <cardExpiration>string</cardExpiration>
        <billingAddress>
          <firstName>string</firstName>
          <lastName>string</lastName>
          <address1>string</address1>
          <address2>string</address2>
          <city>string</city>
          <state>string</state>
          <zip>string</zip>
          <country>string</country>
          <phone>string</phone>
        </billingAddress>
        <shippingAddress>
          <firstName>string</firstName>
          <lastName>string</lastName>
          <address1>string</address1>
          <address2>string</address2>
          <city>string</city>
          <state>string</state>
          <zip>string</zip>
          <country>string</country>
          <phone>string</phone>
        </shippingAddress>
        <lineItems>
          <item>
            <id>string</id>
            <line>int</line>
            <quantity>int</quantity>
            <price>decimal</price>
          </item>
          <item>
```

```
            <id>string</id>
            <line>int</line>
            <quantity>int</quantity>
            <price>decimal</price>
          </item>
        </lineItems>
      </GetOrderResult>
```

There were two Web service testing scenarios. A scenario where the test clients would directly invoke the Web service via a SOAP message sent over HTTP (the Mercury LoadRunner software in this case is making direct SOAP 1.1 requests over HTTP), and a second where the LoadRunner clients would make an HTML/HTTP request on a ASPX or JSP page running in an application server computer 1 which would in turn invoke a remote Web service running on a separate application server computer 2 via SOAP 1.1 over HTTP.

In the first case, the clients would send a SOAP request for a random order of 10,000 possible orders each containing five repeating (5) line items in the returned order object. This test puts high stress on a single application server hosting the Web Service, and is designed to illustrate how well the application server performs as a Web Service host, servicing SOAP requests from thousands of concurrent users.

In the second script, the LoadRunner client uses an HTTP URL on a ASPX or JSP page with a random query string order number between 1 and 10,000 which is the requested order number. The ASPX or JSP page then makes a remote Web Service SOAP request to the second application server hosting the Web Service, getting back a valid order object, and then formatting a portion of the order object (the userID for the order) into the JSP/ASPX page for return back to the client as HTML. This test simulates an enterprise application integration scenario where Web Services are being used to tie together multiple (possibly disparate) backend systems, and is designed to illustrate how well the application server performs as a Web Service client. In this case, the Web Service host machine was configured with 8 CPUs, while the client machine making the remote Web Service request/formatting result into HTML was tested running 2, 4 and 8 CPUs. The results for each scenario are depicted below.

Web Service Benchmark – Direct Web Service Request

Figure 20. *Peak throughput.*

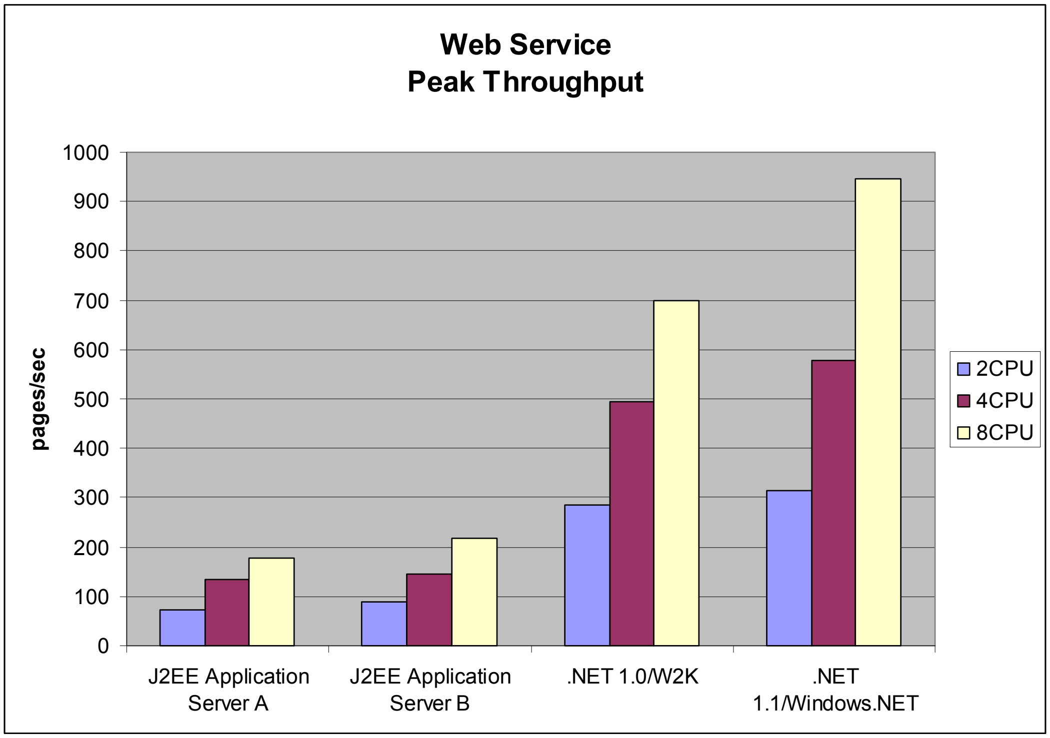


Figure 21. *Maximum supported user load @ 3 second average page response threshold.*

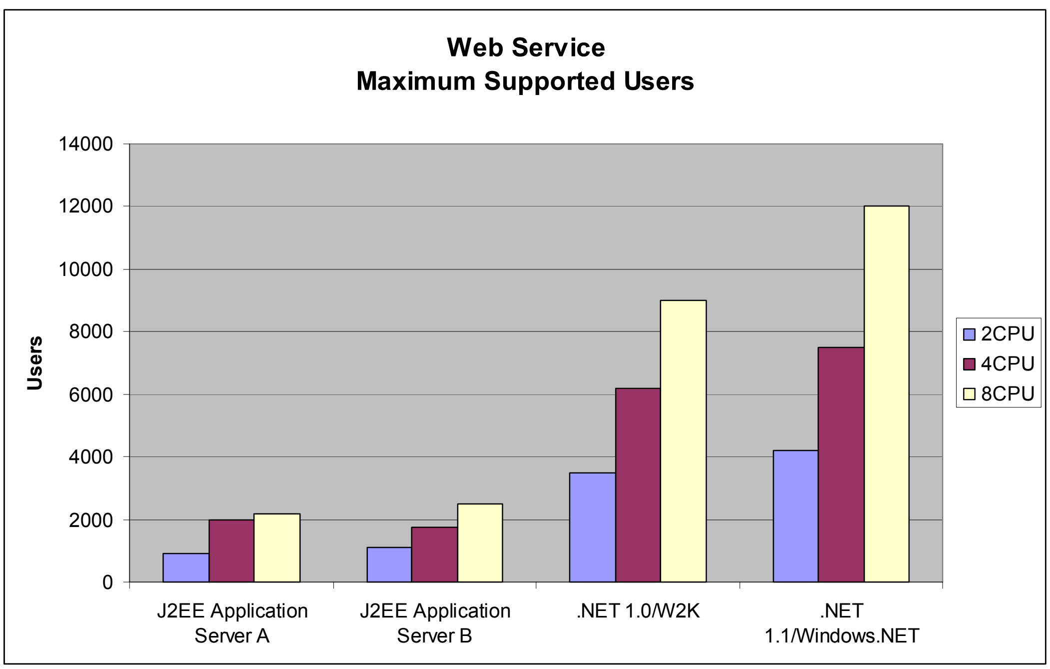


Figure 22. *Throughput curves 2 CPU Application Server.*

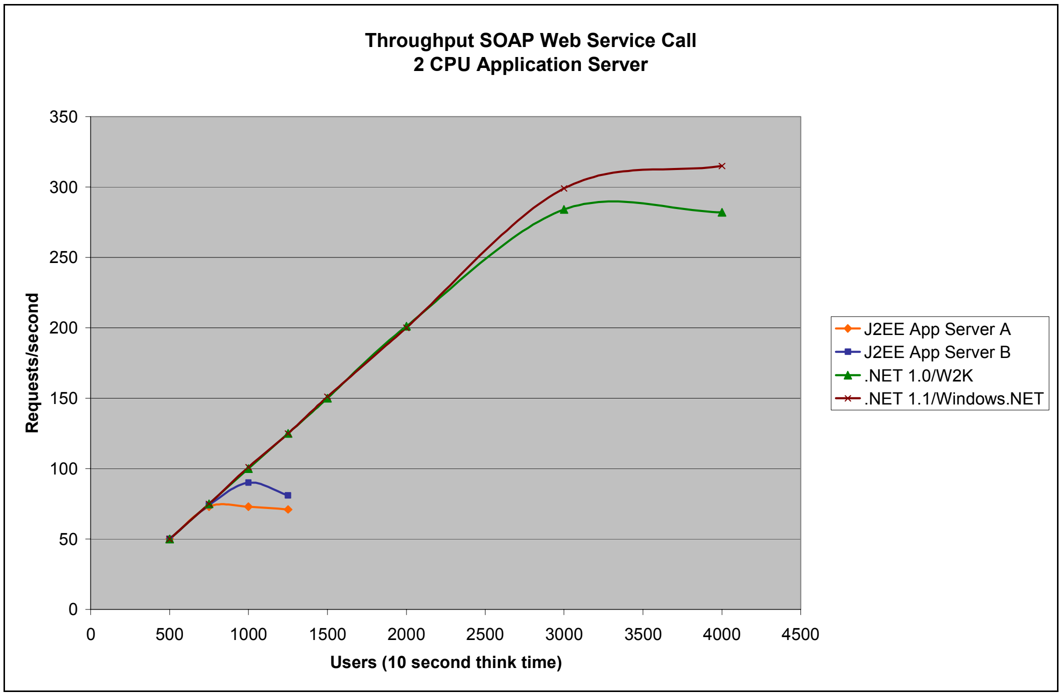


Figure 23. *Throughput curves 4 CPU Application Server.*

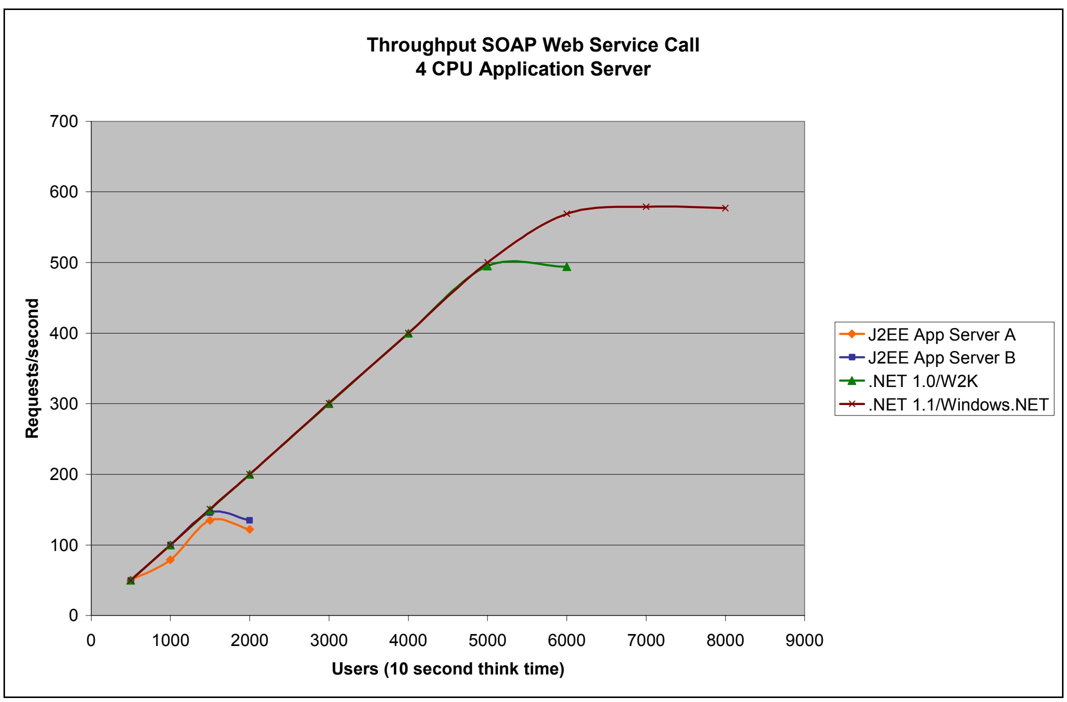
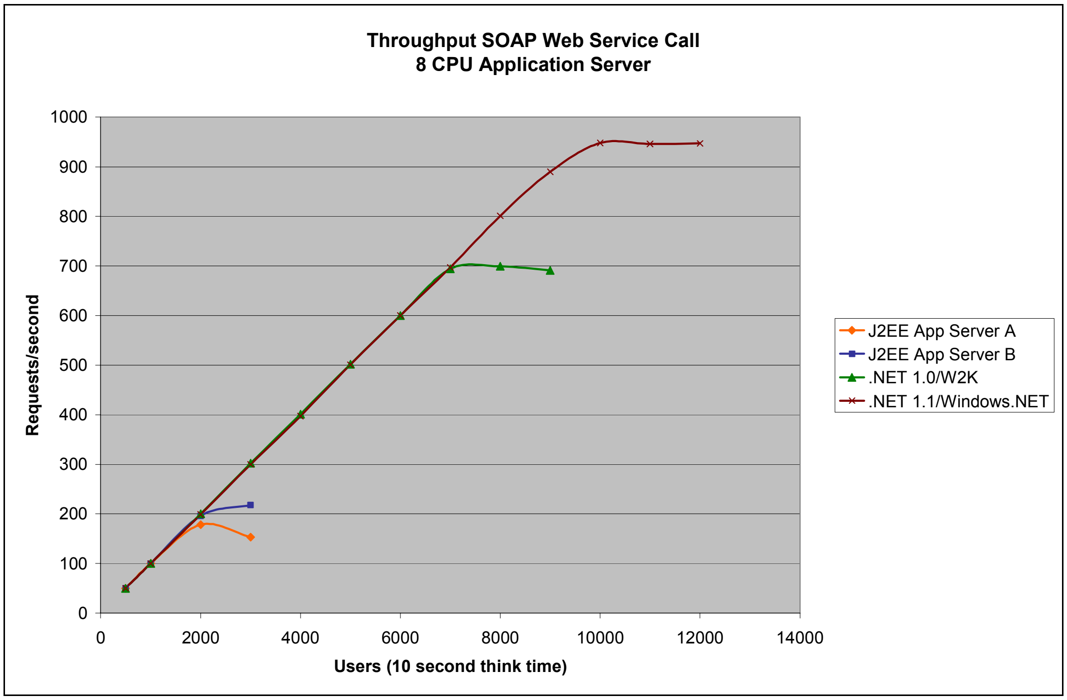
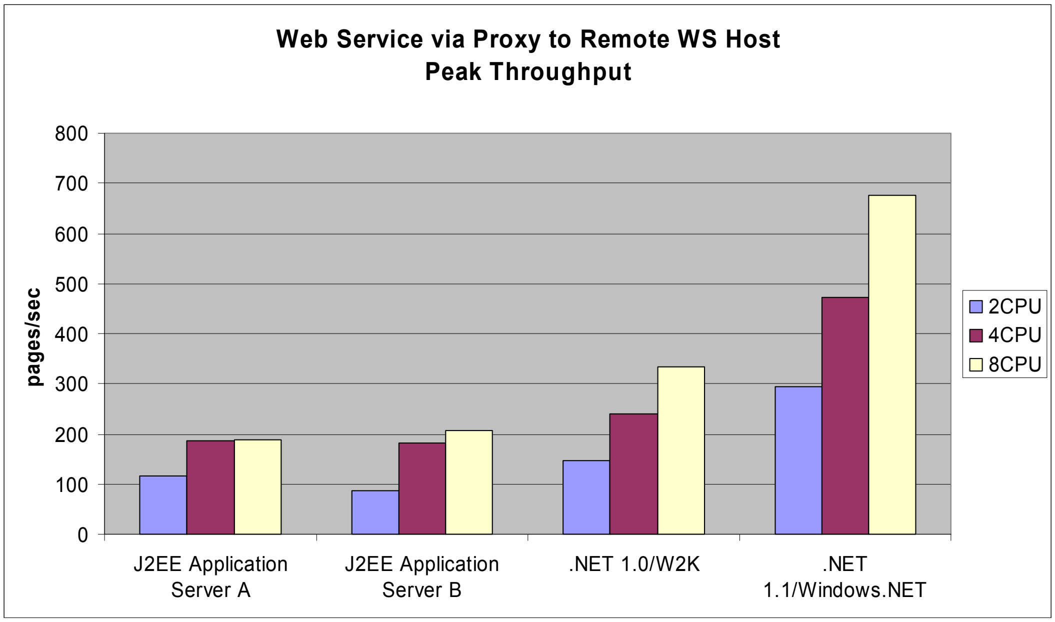


Figure 24. *Throughput curves 8 CPU Application Server.*



Web Service Benchmark – Remote SOAP Client Call

Figure 25. *Peak throughput.*



Observation: On J2EE Application Servers A and B, the 8-CPU Web Service host machine reaches 100% CPU saturation at ~200 TPS, hence little or no additional throughput is achieved on the 8 vs. 4 CPU Web Service client configuration. On Microsoft .NET (both Windows 2000 and Windows.NET Server), the Web Service Host machine does not become saturated before the Web Service Client machine becomes saturated, hence additional throughput is achieved on the 8 vs. 4 CPU Web Service client configuration.

Figure 26. *Maximum supported user load @ 3 second average page response threshold.*

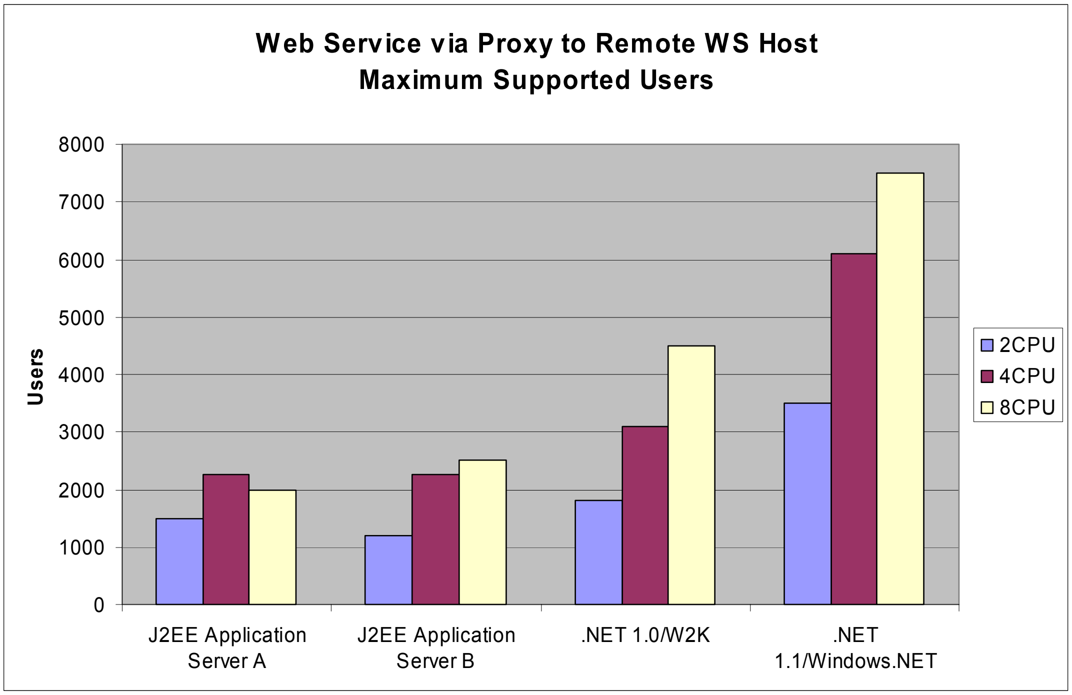


Figure 27. *Throughput curves 2 CPU Application Server.*

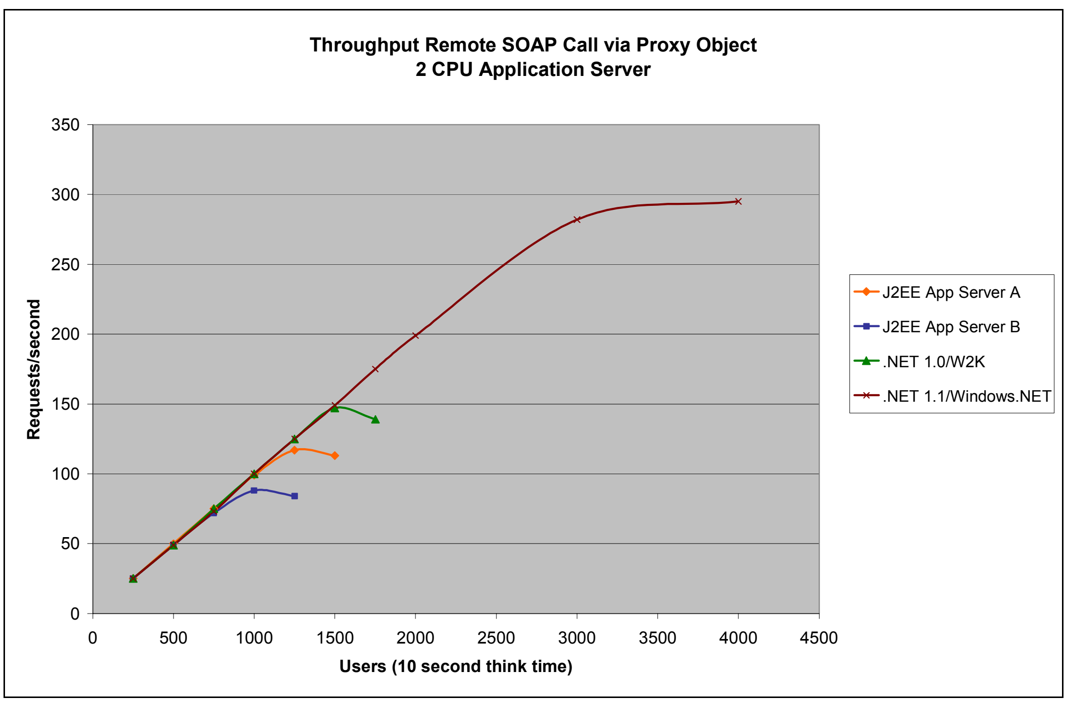


Figure 28. *Throughput curves 4 CPU Application Server.*

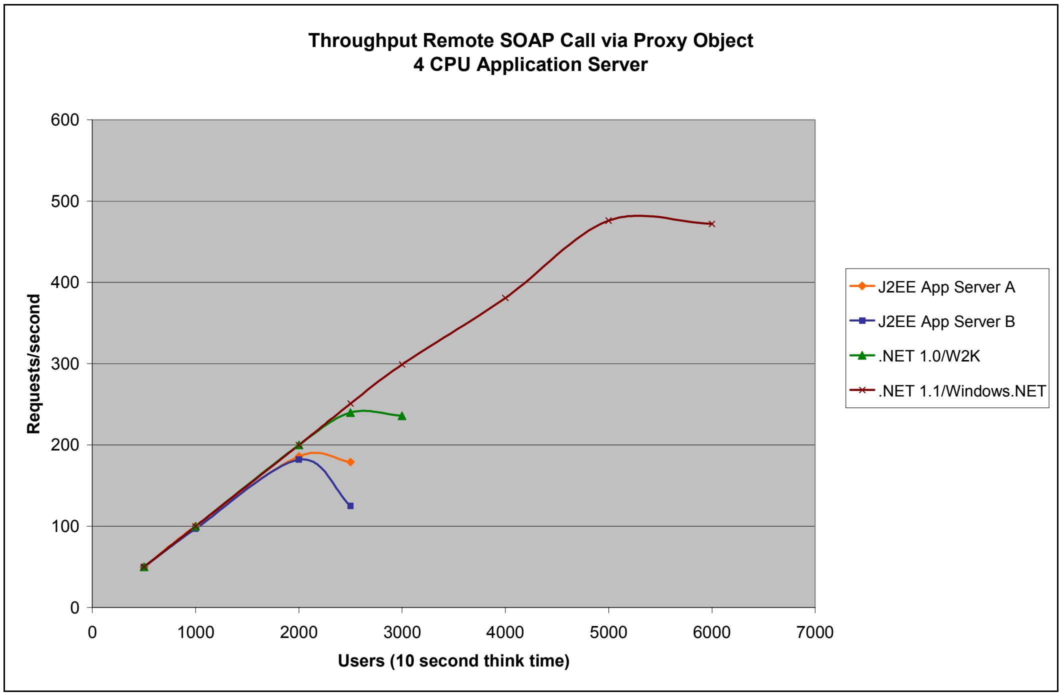
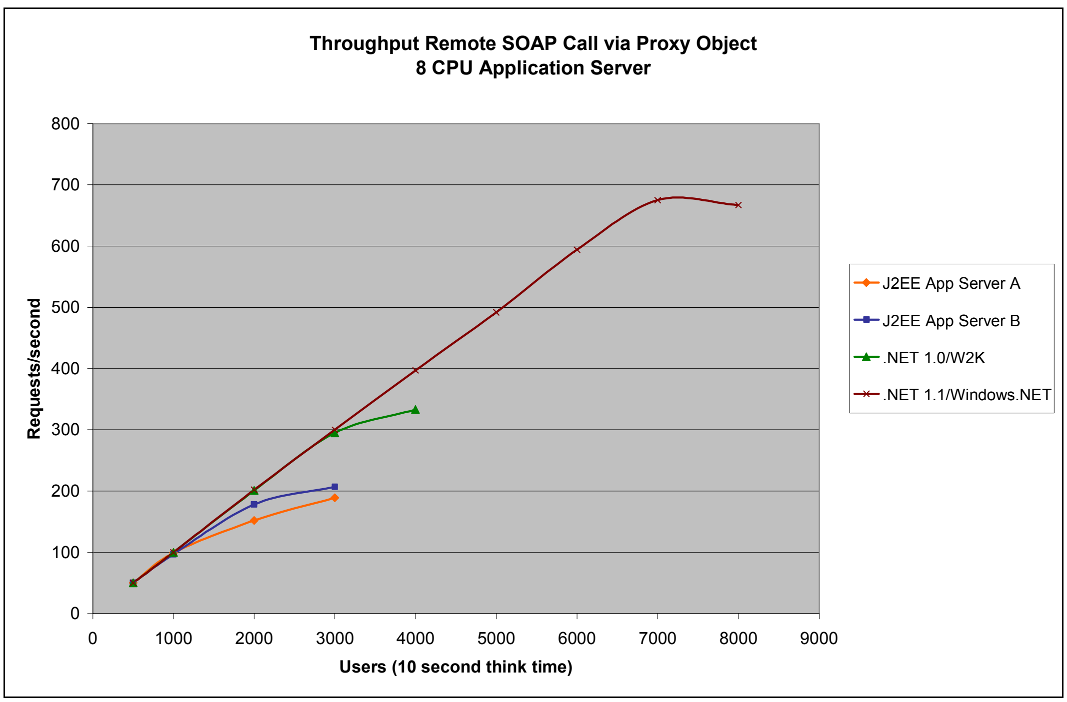


Figure 29. *Throughput curves 8 CPU Application Server.*



Appendix 1- Comparing Lines of Code

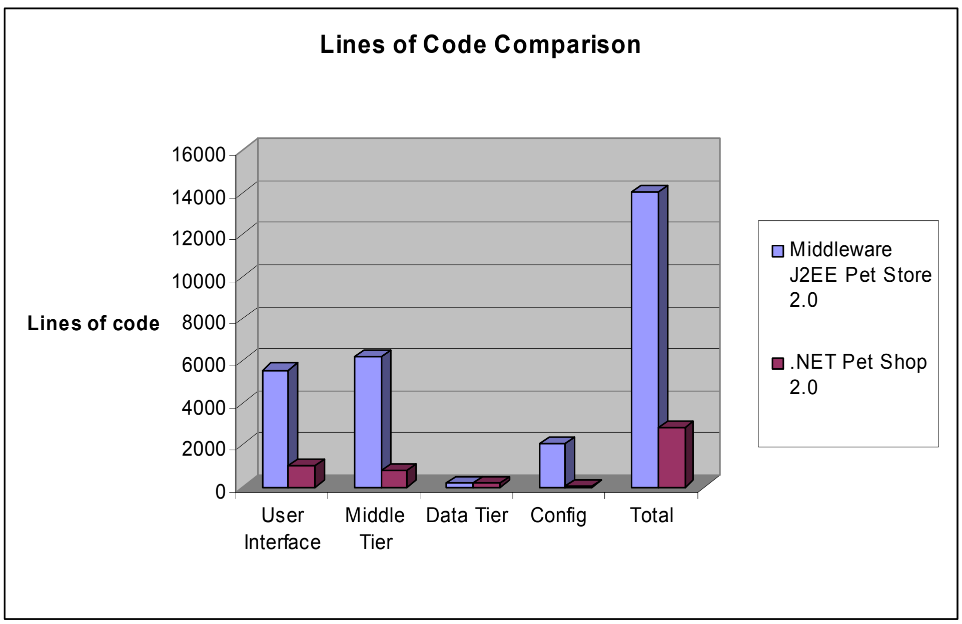
The Middleware Java Pet Store 2.0 implementation uses the same basic EJB-recommended architecture as the original Java Pet Store (except fully optimized for performance). Hence, its code count remains largely unchanged over the original at 14,004 lines of total code.

With the .NET Pet Shop 2.0 implementation, Microsoft has done some further optimizations to reduce its overall line count, while also extending the application with new features for distributed transactions and Web Services. The new .NET Pet Shop 2.0 contains a total of 2,096 lines of C# code (the 1.5 version had a total of 3,484 lines of code, a 40% reduction). Both the .NET and J2EE versions are fully object-oriented, logical n-tier implementations with separation between UI, middle-tier, and data-tier application logic. The complete code count breakdown is depicted below:

Figure 30. *Table depicting lines of code for the different tiers in the implementations.*

Element	.NET Pet Shop 2.0	Middleware J2EE Pet Store 2.0
User Interface	1,002	5,567
Middle Tier	795	6,187
Data Tier	197	197
Configuration	102	2,053
Total	**2,096**	**14,004**

Figure 31. *Comparing the lines of code.*



Appendix 2 – Benchmark Data for Web Application Benchmark

Image Download Off

Throughput Data

Throughput data is measured in average http responses ("hits") served per second by the application server at varying user loads. Note that with image download turned off, this measures the actual number of user requests (pages) per second the server is able to process.

2 CPU Application Server (pgs/sec)

User Load	J2EE App Server A	J2EE App Server B	.NET 1.0/Windows 2000 Server	.NET 1.1/Windows.NET Server
500	53	49	50	50
750	77	68	76	75
1000	106	77	100	100
2000	197		200	200
3000	273		300	300
4000	275		400	398
5000			445	482
6000			450	465

4 CPU Application Server (pgs/sec)

User Load	J2EE App Server A	J2EE App Server B	.NET 1.0/Windows 2000 Server	.NET 1.1/Windows.NET Server
500	50	49	50	50
750	75	71	75	75
1000	100	90	100	100
1500	150	127	150	149
1750	175	135	175	175
1850	185	138	184	185
2000	198	-	200	200
3000	286	-	300	301
4000	385	-	399	402
5000	450	-	500	499
6000	457	-	600	601
7000	-	-	699	701
8000	-	-	789	786
9000	-	-	835	881
10000	-	-	801	891
10500	-	-	812	893
11000	-	-	-	887
12000	-	-	-	883

8 CPU Application Server (pgs/sec)

User Load	J2EE App Server A	J2EE App Server B	.NET 1.0/Windows 2000 Server	.NET 1.1/Windows.NET Server
500	50	49	50	50
750	75	72	75	75
1000	100	93	100	100
1500	151	132	150	150
1750	175	147	174	175
1850	185	161	186	184
2000	200	160	200	200
3000	300	-	300	300
4000	397	-	400	400
5000	498	-	500	500
6000	593	-	600	601
7000	655	-	701	701
8000	633	-	797	795
9000	-	-	896	899
10000	-	-	996	1000
11000	-	-	1091	1102
12000	-	-	1140	1210
13000	-	-	1120	1299
14000	-	-	1101	1394
15000	-	-	1094	1447

Transaction Response Times (seconds)

Note the transaction response times represent a weighted average measured in seconds of all individual per-page response times for the two scenarios, browse and order. This represents how long a user has to wait, on average, for a Web request to complete at varying user loads on the application server.

2 CPU Application Server (seconds)

User Load	J2EE App Server A	J2EE App Server B	.NET 1.0/Windows 2000 Server	.NET 1.1/Windows.NET Server
500	0.012	0.293	0.010	0.008
750	0.013	1.050	0.010	0.009
1000	0.014	3.025	0.010	0.010
2000	0.178	-	0.010	0.013
3000	0.970	-	0.012	0.016
4000	4.550	-	0.020	0.022
5000	-	-	1.237	0.391
6000	-	-	2.952	2.569

4 CPU Application Server (seconds)

User Load	J2EE App Server A	J2EE App Server B	.NET 1.0/Windows 2000 Server	.NET 1.1/Windows.NET Server
500	0.011	0.216	0.010	0.008
750	0.014	0.549	0.010	0.008
1000	0.013	1.100	0.010	0.009
1500	0.014	1.857	0.010	0.009
1750	0.022	2.948	0.010	0.009
1850	0.024	3.444	0.011	0.010
2000	0.060	-	0.010	0.008
3000	0.155	-	0.011	0.008
4000	0.374	-	0.012	0.008
5000	1.102	-	0.013	0.009
6000	3.134	-	0.017	0.011
7000	-	-	0.030	0.017
8000	-	-	0.167	0.025
9000	-	-	0.796	0.240
10000	-		2.500	1.301
10500	-	-	2.913	1.936
11000	-	-	-	2.445
12000	-	-	-	3.625

8 CPU Application Server (seconds)

User Load	J2EE App Server A	J2EE App Server B	.NET 1.0/Windows 2000 Server	.NET 1.1/Windows.NET Server
500	0.010	0.234	0.010	0.007
750	0.011	0.502	0.010	0.007
1000	0.012	0.783	0.010	0.007
1500	0.012	1.418	0.011	0.007
1750	0.011	1.464	0.029	0.007
1850	0.012	1.509	0.031	0.007
2000	0.013	2.487	0.041	0.008
3000	0.013	-	0.011	0.008
4000	0.040	-	0.011	0.008
5000	0.041	-	0.012	0.008
6000	0.129	-	0.151	0.010
7000	0.679	-	0.016	0.010
8000	2.823	-	0.021	0.010
9000	-	-	0.032	0.013
10000	-	-	0.046	0.011
11000	-	-	0.074	0.011
12000	-	-	0.539	0.014
13000	-	-	1.615	0.038
14000	-	-	2.728	0.078
15000	-	-	3.705	0.549

Image Download On

Throughput data is measured in average http responses per second ("hits") served by the application server at varying user loads, inclusive of images returned for each page when images are not yet populated in the browser cache. Because this metric includes images, the hits served per second is higher at a given user load than with image download turned off, however, the servers hit their saturation point sooner as observed by higher transaction response times.

Throughput Data (pages and images served per second)

2 CPU Application Server (pgs and images/sec)

User Load	J2EE App Server A	J2EE App Server B	.NET 1.0/Windows 2000 Server	.NET 1.1/Windows.NET Server
100	37	38	55	57
250	97	96	138	135
500	196	183	275	266
1000	385	153	553	551
2000	641	-	1099	1092
3000	584	-	1108	1634
4000	-	-	976	2112
5000	-	-	-	2043

4 CPU Application Server (pgs and images/sec)

User Load	J2EE App Server A	J2EE App Server B	.NET 1.0/Windows 2000 Server	.NET 1.1/Windows.NET Server
100	38	39	55	56
250	99	94	138	140
500	197	186	277	274
750	276	274	407	411
1000	393	243	551	561
2000	774	-	1101	1020
3000	1113	-	1648	1627
4000	1141	-	2190	2191
5000	1088	-	2734	2721
6000	-	-	2919	3275
7000	-	-	2850	3785
8000	-	-	-	3914
9000	-	-	-	3904

8 CPU Application Server (pgs and images/sec)

User Load	J2EE App Server A	J2EE App Server B	.NET 1.0/Windows 2000 Server	.NET 1.1/Windows.NET Server
100	40	39	55	41
250	94	95	93	145
500	198	187	276	274
750	276	276	408	407
1000	394	332	552	507
1250	488	335	690	703
2000	784	-	1101	1067
3000	1122	-	1647	1630
4000	1441	-	2197	2118
5000	1521	-	2739	2725
6000	1569	-	3293	3307
7000	-	-	3812	3834
8000	-	-	4055	4246
9000	-	-	3892	4902
10000	-	-	-	5481
11000	-	-	-	6011
12000	-	-	-	6360
13000	-	-	-	6883
14000	-	-	-	

Transaction Response Times (seconds)

With image download on, this measures how long each page (request) takes to return inclusive of images being loaded from the server.

2 CPU Application Server (seconds)

User Load	J2EE App Server A	J2EE App Server B	.NET 1.0/Windows 2000 Server	.NET 1.1/Windows.NET Server
100	0.022	0.053	0.010	0.014
250	0.019	0.069	0.011	0.013
500	0.020	0.530	0.012	0.013
1000	0.026	10.878	0.012	0.012
2000	2.115	-	0.015	0.014
3000	10.060	-	0.095	0.024
4000	-	-	5.073	0.357
5000	-	-	-	3.467

4 CPU Application Server (seconds)

User Load	J2EE App Server A	J2EE App Server B	.NET 1.0/Windows 2000 Server	.NET 1.1/Windows.NET Server
100	0.016	0.041	0.013	0.012
250	0.016	0.250	0.012	0.012
500	0.018	0.383	0.011	0.012
750	0.019	0.666	0.011	0.012
1000	0.020	6.184	0.012	0.012
2000	0.100	-	0.012	0.013
3000	0.462	-	0.014	0.013
4000	3.570	-	0.016	0.015
5000	7.685	-	0.037	0.021
6000	-	-	1.312	0.026
7000	-	-	3.527	0.149
8000	-	-	-	1.207
9000	-	-	-	2.739

8 CPU Application Server (seconds)

User Load	J2EE App Server A	J2EE App Server B	.NET 1.0/Windows 2000 Server	.NET 1.1/Windows.NET Server
100	0.016	0.044	0.010	0.012
250	0.016	0.186	0.011	0.014
500	0.018	0.322	0.011	0.012
750	0.020	0.513	0.012	0.012
1000	0.022	1.614	0.012	0.012
1250	0.020	4.349	0.012	0.012
2000	0.028	-	0.013	0.013
3000	0.484	-	0.014	0.013
4000	0.846	-	0.015	0.014
5000	2.846	-	0.019	0.015
6000	5.012	-	0.028	0.017
7000	-	-	0.084	0.019
8000	-	-	0.846	0.020
9000	-	-	2.704	0.027
10000	-	-	-	0.028
11000	-	-	-	0.052
12000	-	-	-	0.373
13000	-	-	-	2.635
14000	-	-	-	

Appendix 3 – Benchmark Data for 24 Hour Distributed Transaction Benchmark

For this benchmark, peak throughput was determined for a 4-CPU application server. The test was then run at this user load for 24 hours. This test shows if the peak throughput is sustainable over time.

Comparative Transaction Performance in TPS and $/TPS

	Peak User Load	Average TPS (orders/sec)	Total Errors	Total Application Server System Cost	Price/ performance in $/tps*
JEEE App Server A	4,000	59	40	$76,990	**$1,305**
J2EE App Server B	1000	18 (not sustained)	100% after 4 hours	$84,990	**$4,722**
Microsoft .NET/Windows 2000 Server	4,000	79	0	$36,990	**$468**
Microsoft .NET 1.1/Windows.NET Server	6,000	117	0	**36,990**	**$316**

*Price performance is calculated by summing the middle-tier licensing costs for the application server inclusive of cost of OS (Windows 2000 Advanced Server + Internet Connector), the cost of the hardware, and cost of application server software for a 4-CPU configuration, broken out in the table below. Cost of database software and ongoing maintenance/support costs are not included.

Estimated Application Server Software + Hardware System Cost

	Cost of Application Server Software[4]	Cost of Operating System[5]	Cost of Hardware[6]	Total Cost of Application Server
JEEE App Server A on Windows 2000 SERVER	$40,000	$5,990	$31,000.00	**$76,990.00**
J2EE App Server B on Windows 2000 Server	$48,000	$5,990	$31,000.00	**$84,990.00**
Microsoft .NET on Windows 2000 Server	$0.00 (License-free download for W2K)	$5,990	$31,000.00	**$36,990.00**
Microsoft .NET 1.1 on Windows.NET Server 2003	$0.00 (integrated into Windows.NET Server 2003	$5,990[7]	$31,000.00	**$36,990.00**

[4] Inclusive of license costs on a per-CPU basis for a 4-CPU configuration of the edition that provides enterprise functionality, including the ability to cluster multiple managed servers.
[5] Cost of OS includes a license for Windows 2000 Advanced Server with 25 client access licenses ($3,995) and an Internet Connector License ($1,995) for unlimited anonymous internet access to that server.
[6] Base cost for a Compaq ProLiant DL760 (replacement model for Compaq 8500) with 4 CPUs and 2 GIG RAM.
[7] Estimate based on Windows 2000 Advanced Server pricing. Pricing for Windows.NET Server 2003 has not been published.

Appendix 4 – Benchmark Data for Web Services Benchmark

Web Service Direct SOAP Activation from Load Generators

Throughput (responses per second)

2 CPU Application Server (responses/sec)

User Load	J2EE App Server A	J2EE App Server B	.NET 1.0/Windows 2000 Server	.NET 1.1/Windows.NET Server
500	50	50	50	50
750	73	74	75	75
1000	73	90	100	101
1250	71	81	125	125
1500	-	-	150	151
2000	-	-	201	200
3000	-	-	284	299
4000	-	-	282	314

4 CPU Application Server (responses/sec)

User Load	J2EE App Server A	J2EE App Server B	.NET 1.0/Windows 2000 Server	.NET 1.1/Windows.NET Server
500	50	50	50	50
1000	79	100	100	100
1500	135	146	150	151
2000	122	135	200	200
3000	-	-	300	301
4000	-	-	400	400
5000	-	-	495	500
6000	-	-	494	569
7000	-	-	-	579
8000	-	-	-	577

8 CPU Application Server (responses/sec)

User Load	J2EE App Server A	J2EE App Server B	.NET 1.0/Windows 2000 Server	.NET 1.1/Windows.NET Server
500	50	50	50	50
1000	100	100	100	100
2000	178	196	200	199
3000	153	218	302	300
4000	-	-	401	397
5000	-	-	502	501
6000	-	-	600	600
7000	-	-	694	697
8000	-	-	699	801
9000	-	-	691	890
10000	-	-	-	948
11000	-	-	-	946
12000	-	-	-	947

Response Times (seconds)

2 CPU Application Server (seconds)

User Load	J2EE App Server A	J2EE App Server B	.NET 1.0/Windows 2000 Server	.NET 1.1/Windows.NET Server
500	0.062	0.069	0.010	0.010
750	0.376	0.167	0.011	0.011
1000	3.849	1.141	0.012	0.012
1250	7.493	5.428	0.012	0.012
1500	-	-	0.013	0.012
2000	-	-	0.017	0.017
3000	-	-	0.609	0.091
4000	-	-	4.209	2.772

4 CPU Application Server (seconds)

User Load	J2EE App Server A	J2EE App Server B	.NET 1.0/Windows 2000 Server	.NET 1.1/Windows.NET Server
500	0.041	0.040	0.010	0.011
1000	0.079	0.074	0.010	0.011
1500	1.210	0.266	0.011	0.012
2000	2.991	4.807	0.012	0.012
3000	-	-	0.014	0.014
4000	-	-	0.018	0.017
5000	-	-	0.116	0.030
6000	-	-	2.192	0.415
7000	-	-	-	2.137
8000	-	-	-	3.885

8 CPU Application Server (seconds)

User Load	J2EE App Server A	J2EE App Server B	.NET 1.0/Windows 2000 Server	.NET 1.1/Windows.NET Server
500	0.037	0.036	0.011	0.011
1000	0.068	0.049	0.011	0.011
2000	1.302	0.187	0.011	0.012
3000	9.688	3.789	0.012	0.013
4000	-	-	0.013	0.014
5000	-	-	0.016	0.015
6000	-	-	0.024	0.018
7000	-	-	0.132	0.062
8000	-	-	1.490	0.030
9000	-	-	3.066	0.060
10000	-	-	-	0.603
11000	-	-	-	1.686
12000	-	-	-	2.712

Web Service Remote SOAP Client Call

Throughput (responses per second)

Responses per second are measured as valid pages returned per second by the Web Service client application server. In order to return a page, the client application server must activate the remote Web Service via its proxy object, wait for a SOAP response, then format that response into HTML before returning the page to the calling user.

2 CPU Application Server (responses/sec)

User Load	J2EE App Server A	J2EE App Server B	.NET 1.0/Windows 2000 Server	.NET 1.1/Windows.NET Server
250	25	25	25	25
500	50	49	49	49
750	75	72	75	73
1000	99	88	100	100
1250	117	84	125	125
1500	113	-	147	149
1750	-	-	139	175
2000	-	-	-	199
3000	-	-	-	282
4000	-	-	-	295

4 CPU Application Server (responses/sec)

User Load	J2EE App Server A	J2EE App Server B	.NET 1.0/Windows 2000 Server	.NET 1.1/Windows.NET Server
500	50	50	50	50
1000	100	97	100	100
2000	186	182	200	200
2500	179	125	240	251
3000	-	-	236	299
4000	-	-	-	381
5000	-	-	-	476
6000	-	-	-	472

8 CPU Application Server (responses/sec)

User Load	J2EE App Server A	J2EE App Server B	.NET 1.0/Windows 2000 Server	.NET 1.1/Windows.NET Server
500	50	50	50	50
1000	99	97	100	100
2000	152	178	201	202
3000	189	207	295	300
4000	-	-	333	397
5000	-	-	-	492
6000	-	-	-	594
7000	-	-	-	675
8000	-	-	-	667

Response Times (seconds)

2 CPU Application Server (seconds)

User Load	J2EE App Server A	J2EE App Server B	.NET 1.0/Windows 2000 Server	.NET 1.1/Windows.NET Server
250	0.048	0.034	0.022	0.018
500	0.056	0.246	0.024	0.019
750	0.071	0.425	0.028	0.023
1000	0.134	1.395	0.035	0.065
1250	0.719	4.844	0.051	0.024
1500	3.286	-	0.291	0.049
1750	-	-	2.64	0.048
2000	-	-	-	0.037
3000	-	-	-	0.377
4000	-	-	-	3.633

4 CPU Application Server (seconds)

User Load	J2EE App Server A	J2EE App Server B	.NET 1.0/Windows 2000 Server	.NET 1.1/Windows.NET Server
500	0.089	0.217	0.023	0.018
1000	0.064	0.261	0.026	0.023
2000	0.86	0.992	0.045	0.031
2500	4.002	10.097	1.7844	0.042
3000	-	-	2.682	0.049
4000	-	-	-	0.542
5000	-	-	-	0.699
6000	-	-	-	2.753

8 CPU Application Server (seconds)

User Load	J2EE App Server A	J2EE App Server B	.NET 1.0/Windows 2000 Server	.NET 1.1/Windows.NET Server
500	0.052	0.213	0.023	0.018
1000	0.066	0.249	0.032	0.018
2000	3.182	1.208	0.049	0.022
3000	5.936	4.48	0.205	0.055
4000	-	-	2.07	0.123
5000	-	-	-	0.229
6000	-	-	-	0.331
7000	-	-	-	1.456
8000	-	-	-	3.585

Appendix 5 – Tuning on J2EE Application Server A

Web Benchmark

JRE – SunSoft 1.4 with options –server –Xms350m –Xmx350m
-XX:CompileThreshold=3500 and -Xconcgc

JDBC for non XA – Oracle 9.2 thick drivers
JDBC for XA – Application server vendor thick Oracle driver

Number of application server threads 7 for 2 processors, 5 for 4 processors and 4 for 8 processors

For each processor level the number of running server instances was equal to the number of processors. Each server instance ran in an independent mode as configured above, and listened on an individual IP address addressable via DNS on the clients.

Distributed Transaction Benchmark

Same as Web benchmark above.

Web Service Benchmark

JSP (Web service client) servers:

JRE – SunSoft 1.4 with options –Xms200m –Xmx200m and –Xconcgc
Number of server threads 10

<u>WebService Server</u>

JRE – JRockit 3.1.5 with options –client –Xms400m –Xmx400m, note: SunSoft 1.4 was incompatible with the application server Web service API
Number of server threads 4
JDBC – Oracle 9.2 thick drivers

For each processor level the number of JSP servers running was equal to the number of servers running the Web service. These numbers were: 2 of each for 2 processors and 4 of each for 4 and 8 processors. . Each server instance ran in an independent mode as configured above, and listened on an individual IP address addressable via DNS on the clients.

Appendix 6 – Tuning on J2EE Application Server B

Web Benchmark

JRE – IBM 1.3 with options –server –Xms512m –Xmx512m

JDBC for non XA – Oracle 9.2 thin drivers
JDBC for XA – Oracle 9.2 thin drivers

Number of application server threads 22 for all processor levels

Number of Web server threads was set at a maximum of 50 for all processor levels

For all processor levels the number of running application server instances was two
For all processor levels only one instance of the front end Web server was used. The Web Server used ships as part of the application server tested, and hence is part of the packaged product.

Distributed Transaction Benchmark

Same as Web benchmark above.

Web Service Benchmark

JSP (Web service client) servers:

JRE – IBM 1.3 with options –server –Xms512m –Xmx512m
Number of server threads 22

WebService Server

JRE – IBM 1.3 with options –server –Xms512m –Xmx512m
Number of server threads 22
JDBC – Oracle 9.2 thin drivers

For each processor level the number of JSP servers running was two.

Appendix 7 – Tuning on .NET 1.0/Windows 2000 Server

The following configuration changes were made.

Basic global configuration changes:

- Setting IIS 5.0 authentication, turning off Windows integrated, allowing anonymous user access to the Pet Shop virtual directory (anonymous users are authenticated on check-in via .NET forms-based authentication).
- Ensuring IIS performance tab set to >100,000 hits per day.
- Pointing the COM+ local distributed transaction coordinator (DTC) on the application server and the Products database server to utilize the DTC on the Orders database for distributed transaction logging purposes.
- Ensuring "autoeventwireup=true" set at top of ASPX pages.

Web Benchmark

```
Machine.config file changes:

<httpRuntime

.

.

appRequestQueueLimit="4000"/>    //note this simply adjusts number of
queued requests allowed before server returns "server busy" error.  It
should be adjusted based on number of procs in system, and how long you
want users to wait for returned pages vs. returning a too busy error
message.

<authentication mode="Forms">

<processModel enable="False"

.

.

maxWorkerThreads="20"

maxIoThreads="20"/>
```

Distributed Transaction Benchmark

```
Machine.config file changes:

<authentication mode="Forms">

<httpRuntime

.

.

appRequestQueueLimit="4000"/>

<processModel enable="False"
```

```
.
.
maxWorkerThreads="12"
maxIOThreads="12"/>
```

Web Service Benchmark

```
Machine.config file:

<connectionManagement>
<add address="*" maxconnection="32"/>
</connectionManagement>
<httpRuntime
.
.
appRequestQueueLimit="4000"/>
minFreeThreads="32"
minLocalRequestFreeThreads="16" />

<authentication mode="Forms">
<processModel enable="False"
.
.
maxWorkerThreads="16"
maxIOThreads="16"/>
```

Appendix 8– Tuning on .NET 1.1/Windows.NET Server 2003

The following configuration changes were made. Note these settings apply to release candidates of Windows.NET Server 2003 and the .NET Framework 1.1, and may differ on final released product.

Basic global configuration changes:

- Setting IIS 6.0 authentication, turning off Windows integrated authentication, allowing anonymous user access to the Pet Shop virtual directory (anonymous users are authenticated on check-in via .NET forms-based authentication).
- Pointing the COM+ local distributed transaction coordinator (DTC) on the application server and the Products database server to utilize the DTC on the Orders database for distributed transaction logging purposes.
- Ensuring “autoeventwireup=true” set at top of ASPX pages.
- Adding registry key [HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\Http\Parameters\MaxConnections] with value 40000 (this is throttled by default to help prevent denial of service attacks, but for high concurrent users or benchmark scenarios, should be adjusted upwards)

Web Benchmark

```
Two Web sites created (pointing to same application files), each
listening on one subnet/IP.

Two application pools created, one for each Web site.

This results in two worker processes for the application.

Machine.config file changes:

<httpRuntime

.

.

appRequestQueueLimit="4000"/>


<authentication mode="Forms">
```

Distributed Transaction Benchmark

```
Same as Web Benchmark.
```

Web Service Benchmark

Web Service Host

```
2 Worker Processes in the default application pool.

Machine.config file changes:

<connectionManagement>
<add address="*" maxconnection="32"/>
</connectionManagement>
```

```
<httpRuntime
.
.
appRequestQueueLimit="4000"
minFreeThreads="32"
minLocalRequestFreeThreads="16" />

<authentication mode="Forms">
```

Web Service Client

Two Web sites created (pointing to same application files), each listening on one subnet/IP.

Two application pools created, one for each Web site.

This results in two worker processes for the application.

Machine.config file changes:

```
<connectionManagement>
<add address="*" maxconnection="32"/>
</connectionManagement>
<httpRuntime
.
.
appRequestQueueLimit="4000"
minFreeThreads="32"
minLocalRequestFreeThreads="16" />

<authentication mode="Forms">
```